

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

 MANNING

官方团队编写 NoSQL排名第一

MongoDB IN ACTION
Second Edition

MongoDB

实战

(第二版)

Kyle Banker

Peter Bakkum

[美] Shaun Verch 著

Douglas Garrett

Tim Hawkin

徐雷 徐扬 译



华中科技大学出版社

<http://www.hustp.com>

内容提要

第二版

MongoDB 实战

Kyle Banker

Peter Bakkum

[美] Shaun Verch 著

Douglas Garrett

Tim Hawkins

徐雷 徐扬 译

责任编辑：王

封面设计：王

版式设计：王

校对：王

印数：10000册

（京）登字：0000000

4 本书的原创

辛 880 元

我学习技术开发的原则就是广泛阅读国外经典图书。其次就是阅读官方文档。本书就是我在开发MongoDB的过程中，结合自己的经验，将MongoDB的官方文档进行整理和总结，并加入自己的理解和实践经验，编写而成的。本书旨在帮助读者快速掌握MongoDB的实战应用，提高开发效率。

华中科技大学出版社

中国·武汉



内 容 提 要

本书分三部分通过大量的实例代码介绍了 MongoDB 数据库底层的实现以及大型互联网 Web 项目数据库设计原则。第一部分对 MongoDB 进行了整体介绍,并介绍了实际的开发例子,另外还介绍了 JavaScript shell 和 Ruby 驱动。第二部分通过逐步实现一个电商数据模型和实现必要的 CRUD 操作来详细介绍了 MongoDB 的文档数据模型、查询语言和 CRUD(新增、读取、更新和删除)操作。本书的最后部分从数据库专家的角度来看待 MongoDB,介绍了数据库的性能、部署、容错和伸缩性等所有的知识。

本书适合想深入学习 MongoDB 的开发人员,主要关注 MongoDB 数据库。

Original English language edition published by Manning Publications, USA. Copyright © 2015 by Manning Publications. Simplified Chinese-language edition copyright © 2016 by HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY PRESS. All rights reserved.

湖北省版权局著作权合同登记 图字:17-2017-015 号

图书在版编目(CIP)数据

MongoDB 实战:第二版/[美]科勒·班克等著;徐雷,徐杨译. —武汉:华中科技大学出版社,2017.3
ISBN 978-7-5680-2579-9

I. ①W… II. ①科… ②徐… ③徐… III. ①关系数据库系统 IV. ①TP311.138

中国版本图书馆 CIP 数据核字(2017)第 031379 号

[美]Kyle Banker, Peter Bakkum, Shaun Verch 著
Douglas Garrett, Tim Hawkins

MongoDB 实战(第二版)

MongoDB Shizhan

徐雷 徐杨 译

策划编辑:谢燕群

责任编辑:谢燕群

责任校对:李 琴

责任监印:周治超

出版发行:华中科技大学出版社(中国·武汉)

电话:(027)81321913

武汉市东湖新技术开发区华工科技园

邮编:430223

录 排:华中科技大学惠友文印中心

印 刷:湖北新华印务有限公司

开 本:787mm×960mm 1/16

印 张:26

字 数:660 千字

版 次:2017 年 3 月第 2 版第 1 次印刷

定 价:80.00 元



华中科技

本书若有印装质量问题,请向出版社营销中心调换
全国免费服务热线:400-6679-118 竭诚为您服务
版权所有 侵权必究

译者序

0. MongoDB 的优点与其学习重要性

MongoDB是最流行的NoSQL数据库！MongoDB在NoSQL中排名第一！它高性能、轻量级，易于扩展，适用于移动互联网敏捷发展，是架构师的必备！应用MongoDB现在也成为Google、Facebook、阿里巴巴、腾讯、百度等互联网公司招聘的关键技能！

1. MongoDB 在 NoSQL 中排名第一

非常荣幸可以翻译MongoDB官方团队撰写的《MongoDB in Action》第二版，这是MongoDB官方团队Kyle Banker和Shaun Verch等精心撰写的图书。此书是MongoDB领域最权威的图书，也是曼宁（Manning）出版社的“××× in Action”系列经典图书之一。本书受到全球读者的五星好评。中文版预计于2017年2月与中国读者见面！

2. 应用 MongoDB 的大型互联网公司

应用MongoDB的国外著名公司有：Google、Facebook、Tweet、思科、Bosch、Adobe、SAP等，国内著名公司有：阿里巴巴、腾讯、百度、360公司、新浪微博、京东、携程、滴滴打车等。新版MongoDB 3.X更是对云计算和大数据分析提供更强的支持，大数据工程师必须掌握MongoDB知识，其重要性不言而喻。

3. 本书优势

《MongoDB实战》（第二版）适合想深入学习MongoDB知识的开发人员，是架构师的必备。本书涵盖了最新版本内容，深入浅出、通过大量的实例代码介绍了MongoDB数据库底层的实现。比如，MongoDB的高可用集群、动态Schema、多服务器分布式数据存储。

4. 我选书的原则

我学习技术开发的原则就是坚持阅读国外经典图书，其次就是阅读官方文档。这个习惯让我受益匪浅。本书就是我们学习MongoDB高级开发的经典图书。本书适合MongoDB开发人员、架构师、DBA、大数据工程师学习！

5. 架构师必须能够对技术难点攻关

架构师必须写代码，必须能够进行技术攻关！作为核心骨干，最大的忌讳就是喜欢糊弄、忽悠，纸上谈兵。我比较反对一些观点：“30岁以后不适合搞技术，架构师不应该写代码”。技术团队的老大不懂技术怎么能带队伍！浮躁的领导会带坏整个公司的技术团队。只有理解框架底层的工作原理，才能在实际工作中分析和优化代码。

6. 致谢与反馈

在业余时间翻译本书的过程中，我还实战MongoDB、编写代码、验证本书中的知识点，比如高可用集群、安全等知识点；在Windows和Linux平台上做了实战部署；作为企业架构顾问参与设计架构方案。本人在翻译本书的过程中，收获很多，重新认识了MongoDB数据库，包括其底层的实现细节。感谢徐扬帮助翻译了部分章节，也要特别感谢华中科技大学出版社编辑们的辛苦工作，是他们才使得本书的翻译得以顺利完成并按期与读者见面。本书是新青年软件训练营（54peixun.com）高级架构班的推荐教材，希望本书能帮助大家深入学习MongoDB。

翻译本书过程中难免出现疏漏错误，如有反馈建议，欢迎加入中国MongoDB开发交流QQ群：511943641，或者微博@徐雷FrankXuLei联系我，非常感谢！

徐雷FrankXuLei

2017年1月18日

- 1) 新青年软件训练营（54peixun.com）联合创始人；
- 2) 微软中国MSDN特邀讲师、微软美国Channel9首位中国讲师；
- 3) 微软大企业客户技术顾问，曾任：购酒网、元拓商城、上海市公安局ERP系统北极绒技术架构顾问；
- 4) 获得吉林大学计算机科学与技术学士学位、上海交通大学硕士学位；
- 5) 翻译国外经典图书《WCF技术内幕》《WCF服务编程》第三版和第四版、《ASP.NET MVC4 Web编程》《jQuery实战》（第三版）、《MongoDB实战》（第二版）；
- 6) 受邀为微软中国、盛大网络、玫琳凯中国研发中心、世界500强约翰迪尔、一嗨租车、沪江网、中国东方航空、美国IGT、世界500强花旗银行、达丰集团、台达集团、美国国家仪器、上海交通大学软件学院、奥伯特石油、中国体彩集团总部、南方电网集团、阿里巴巴恒生证券、诺和诺德集团等中外名企、名校授课；
- 7) 2014年9月16日，受邀作为微软中国开发者技术代表会见ASP.NET之父Scott Gu先生，并作英文报告；
- 8) 为苍老师忠实粉丝，授课幽默风趣，追求“德艺双馨”。

更快节奏的现代生活。

人们感受不同历史时期的文化，正如这本书一样。

0 前言与致谢

在业余时间翻译本书的过程中，我还完成MongoDB、编写代码、验证本书中的知识点，比如 Kyle Banker在10gen连续多年参与MongoDB Ruby驱动开发，之后进入创业公司工作。他写了许多关于MongoDB的技术博客（<http://kylebanker.com/blog>）。

Peter Bakkum使用MongoDB构建可伸缩的大型商业应用系统基础架构领域的专家。

Shaun Verch是就职于MongoDB核心服务器团队，参与研发，并在M101JS讲授一些关于Node.js使用MongoDB开发的课程。

Douglas Garrett是NYC首届全球MongoDB技术大会的演讲嘉宾，MongoDB数据分析创新大奖的获得者，就职于Genentech的生物信息学部门。

Tim Hawkins是Yahoo欧洲搜索引擎的团队“老大”，专注于国际化与可伸缩性领域的解决方案架构研究。

致谢

Acknowledgments

感谢曼宁出版社朋友的帮助，使得本书可以顺利出版。Michael Stephens帮助策划了本书的第一版，对第二版的编辑Susan Conant、Jeff Bleiel、Maureen Spencer在此一并感谢。

写作是个非常耗时的工作。如果没有Eliot Horowitz（MongoDB 公司的CTO）和Dwight Merriman（MongoDB 公司的主席）的慷慨帮助，我可能也无法完成本书。凭借Eliot和Dwight的热情和聪明才智，他们创建了MongoDB，而且他们信任我，让我来参与文档的编写工作。非常感谢他们。

本书中的许多观点收集于我们与10gen公司员工的谈话。因此特别感谢Mike Dirolf、Scott Hernandez、Alvin Richards、Mathias Stearn，还要特别感谢Kristina Chowdorow、Richard Kreuter、Aaron Staple。他们参与了本书第一版许多章节的审核工作，提出了许多专业性的修改建议。

在本书第一版编写阶段，感谢以下朋友参与了初稿的校订工作：Kevin Jackson、Hardy Ferentschik、David Sinclair、Chris Chandler、John Nunemaker、Robert Hanson、Alberto Lerner、Rick Wagner、Ryan Cox、Andy Brudtkuhl、Daniel Bretoi、Greg Donald、Sean Reilly、Curtis Miller、Sanchet Dighe、Philip Hallstrom、Andy Dingley。还要感谢参与本书第二版审核的朋友：Agustin Treceno、Basheeruddin Ahmed、Gavin Whyte、George Girton、Gregor Zurowski、Hardy Ferentschik、Hernan Garcia、Jeet Marwah、Johan Mattisson、Jonathan Thoms、Julia Varigina、Jürgen Hoffmann、Mike Frey、Phlippie Smith、Scott Lyons、Steve Johnson。

特别感谢Wouter Thielen对第10章内容所做的工作，以及技术编辑Mihalis Tsoukalos、参与技术审阅的Doug Warren。

在本书第二版编写期间，我美丽贤惠的妻子Dominika给予了最大的耐心和支持，还有我帅气的儿子Oliver，他非常乖巧听话。非常感谢家人的支持。

Kyle Banker

关于序言

about the Preface

数据库是信息时代的基石。与Atlas(微软1998年发明的Ajax技术)很像,它们默默无闻地支持着我们使用的数字世界。一定要记住,我们的数字交互从评论到推特(美国的微博)搜索和排序,本质上都是与数据库交互。

这种基本的但是隐藏的功能,让我通常对数据库保留一份敬意,与我们穿过悬索桥的感觉不同,通常我们会对汽车保留一份敬畏。

数据库有许多种形式。书籍的索引以及目录卡片都是数据库排序的一种形式,正如为过去Perl程序员设计的特别结构化文本文件。可能现在最著名的就是复杂的关系型数据库,它们构成了当今世界软件的基础。这些关系型数据库,以及它们的第三范式和SQL接口,仍然稳定地运行着。

在从事Web开发工作几年后,我就迫切想寻找关系型数据库的替代品。当我们看到MongoDB以后,真是欣喜若狂。我喜欢用类JSON的数据结构来表示数据的想法。JSON非常简单、直观,而且易于阅读。MongoDB也使用了JSON语言来构建查询,利于新数据库的使用和推广。高级特性如易于复制和分片使得MongoDB更加强大。到目前为止,我已经基于MongoDB构建了一些应用,体验到它的简便性,我已经彻底“路转粉”爱上它了。

好事多磨。经历许多坎坷,我加入了10gen,一家专门领导开源数据库开发的公司。这两年间,我参与了许多客户端驱动的开发和改进工作,与许多客户公司合作。这些经历让我收获颇多,我也希望把一些宝贵的经验在本书中与大家分享。

作为持续不断开发的软件产品,MongoDB仍然不够完美,还在持续完善。但是,MongoDB集群已经成功支持了成千上万的大小小应用,而且日益成熟。许多开发者已经听说过它的强大之处,不仅可以创造奇迹,而且能够带来快乐。希望它也可以改变你的工作。

这是《MongoDB实战》的第二版,我希望你能享受阅读与学习本书的过程。

Kyle Banker

【注1】安装MongoDB驱动可以使用包管理工具,如Ubuntu的apt-get,Debian的dpkg,Redhat的yum,Mac的brew,十分方便。安装驱动,请参考[http://www.mongodb.org/downloads](#)。

关于本书

about this book

本书适合从零开始深入学习MongoDB的开发者和DBA。如果你是个新手，你会发现本书的节奏很好，深入浅出。如果你是有经验的开发者，本书也提供了针对高级知识更加详细的参考。关于本书的内容深度，本书应该适合绝大多数高级开发者。虽然本书内容针对最新的MongoDB版本（当前已经是3.0.x版本），但是也涵盖了之前的MongoDB版本（比如2.6版本）内容。

例子代码由JavaScript、MongoDB shell的语言以及另外一种流行的脚本语言Ruby编写。每次努力都是为了提供更加简单有用的例子，而且仅仅使用了JavaScript和Ruby语言最朴素的特性。主要的目的还是展示MongoDB API最常见的使用方式。如果你已经使用过其他语言，就应该会发现这些例子很容易理解。

关于语言，再强调一点。如果你好奇“为什么不使用那个×语言”，那么你可以放心，官方支持的MongoDB驱动具备一致的功能和API。这意味着只要学习一个语言的API，其他语言的就可以融会贯通了。

如何使用本书

How to use this book

本书部分是教程，部分是参考资料。如果你完全是个MongoDB“菜鸟”，最好还是按照顺序阅读，这样收获最大。本书包含了许多例子代码，可以帮助大家来巩固概念。不过要求大家安装MongoDB和选装Ruby驱动。附录A里有安装指南^[1]。

如果你已经学习过MongoDB，那么可以阅读专门的高级主题。第8~13章以及其余的附录都是独立的知识点，可以任意顺序阅读。此外，第4~7章包含了基本要点，主要关注MongoDB的核心概念。这些内容也可以独立学习。

^[1]【译注】安装MongoDB驱动可以使用包管理工具，.NET有NuGet，Node.js有NPM，十分方便。实在不行，还可以从官网下载驱动包手动加入项目的代码中。

路线图

Roadmap

本书分为三大部分。

第一部分从零开始完整介绍MongoDB。第1章介绍了MongoDB的历史、特性以及用例。第2章通过MongoDB命令shell教会大家核心的概念。第3章介绍如何使用MongoDB设计简单的应用后台数据库。

第二部分详细介绍了第一部分使用的MongoDB API。分4章内容逐步介绍了电商应用schema和操作。第4章深入介绍了文档document，MongoDB里存储数据的最小单元，而且介绍了基本的电商schema设计的例子。第5~7章介绍如何通过查询和更新来使用schema。为了更加细化内容，每章都包含一个更详细的子主题。

第三部分主要是关于MongoDB的高级主题。第8章介绍了索引机制和查询优化的内容。第9章介绍了MongoDB内部的文本搜索引擎。

第10章是本书中的新增内容，关于WiredTiger存储引擎和可拔插存储，这是MongoDB v3之后独有的特性。第11章关注复制，使用MongoDB 高可用策略以及伸缩性的内容。第12章介绍了分片sharding存储、MongoDB的水平伸缩特性。

第13章提供了MongoDB安装、部署、管理和解决错误的最佳实践经验。

本书还包含3个附录部分。附录A介绍了如何在Linux、Mac OS X、Windows下安装MongoDB和Ruby驱动。附录B介绍了一系列schema和设计模式，还有一些反模式列表。附录C介绍了如何在MongoDB存储二进制数据，以及如何使用GridFS，所有驱动都实现同一个规范，这个规范是专门用来存储大文件的技术。

代码约定和下载

Code conventions and downloads

所有的列表代码以及文本代码都使用了固定宽度的字体，以区分其他文字。

对于列表或者高亮的概念会加设代码注释。

有时候，会在文本里使用编号的链接来解释概念。

作为一个开源项目，10gen负责处理社区反馈的所有MongoDB的bug。本书的几个地方，特别是脚注部分，你会参考bug包含的参考建议，以及计划的改进方案。例如，在SERVER-380增加全文搜索功能的申请票据。要查询任意票据的状态，可以浏览<http://jira.mongodb.org>，然后输入ID搜索。

大家也可以从<http://mongodb-book.com>或者<http://manning.com/MongoDBinAction>下载本书的源码，包括一些例子数据。

软件需求

在开始学习本书之前，需要安装MongoDB。

MongoDB的安装指南可以参考附录A以及MongoDB官方网站(<http://mongodb.org>)。

如果要运行Ruby驱动的例子，需要安装Ruby。请参考附录A的安装指南。

第一部分 入门 1

第1章 全新 Web 数据库 3

1.1 为什么要用 4

1.2 MongoDB 简介 5

1.2.1 文档数据库 5

1.2.2 schema 支持 9

1.2.3 索引 9

1.2.4 复制 10

1.2.5 数据分片 11

1.2.6 部署 13

1.3 核心服务和工具 14

1.3.1 核心服务 14

1.3.2 JavaScript 驱动 15

1.3.3 管理控制台 15

1.3.4 命令行工具 16

1.4 为什么是 MongoDB? 17

1.4.1 MongoDB 和其他数据库 17

1.4.2 使用场景和部署 20

1.5 提示和引用 21

1.6 MongoDB 历史 22

1.7 其他资源 23

1.8 总结 25

第2章 通过 JavaScript shell 操作 MongoDB 27

2.1 Drive into the JavaScript MongoDB shell 28

2.1.1 启动 shell 28

2.1.2 帮助命令、命令的文档 29

2.1.3 插入和查询 29

2.1.4 更新文档 31

2.1.5 删除数据 32

目录

Table of Contents

第一部分 入门.....	1
第 1 章 全新 Web 数据库.....	3
1.1 为互联网而生.....	5
1.2 MongoDB 键特性.....	5
1.2.1 文档数据模型.....	5
1.2.2 ad hoc 查询.....	9
1.2.3 索引.....	9
1.2.4 复制.....	10
1.2.5 加速与持久化.....	11
1.2.6 伸缩.....	13
1.3 核心服务和工具.....	14
1.3.1 核心服务器.....	14
1.3.2 JavaScript shell.....	15
1.3.3 数据库驱动.....	15
1.3.4 命令行工具.....	16
1.4 为什么是 MongoDB?.....	17
1.4.1 MongoDB 与其他数据库对比.....	17
1.4.2 使用场景和部署.....	20
1.5 提示和限制.....	22
1.6 MongoDB 历史.....	23
1.7 其他资源.....	25
1.8 总结.....	25
第 2 章 通过 JavaScript shell 操作 MongoDB.....	27
2.1 Diving into the 深入 MongoDB shell.....	28
2.1.1 启动 shell.....	28
2.1.2 数据库、集合和文档.....	28
2.1.3 插入和查询.....	29
2.1.4 更新文档.....	31
2.1.5 删除数据.....	35

2.1.6 shell 的其他特性	35
2.2 使用索引创建和查询	36
2.2.1 创建大集合	36
2.2.2 索引和 explain()	38
2.3 基本管理	42
2.3.1 获取数据库信息	43
2.3.2 命令如何执行	44
2.4 获取帮助	45
2.5 总结	47
第 3 章 编写代码操作 MongoDB	48
3.1 通过 Ruby lens 连接 MongoDB	49
3.1.1 安装与连接	49
3.1.2 Ruby 里插入文档数据	50
3.1.3 查询与光标	51
3.1.4 更新和删除	52
3.1.5 数据库命令	53
3.2 驱动工作原理	54
3.3 构建简单的应用	56
3.3.1 设置	56
3.3.2 搜集数据	57
3.3.3 查看存档	60
3.4 总结	63
第二部分 MongoDB 应用系统开发	65
第 4 章 面向文档的数据	67
4.1 schema 设计原则	67
4.2 设计电商网站数据模型	69
4.2.1 schema 基础知识	69
4.2.2 用户和订单	73
4.2.3 评价	75
4.3 核心概念: 数据库、集合、文档	76
4.3.1 数据库	76
4.3.2 集合	79
4.3.3 文档和插入	83
4.4 总结	87
第 5 章 构建查询	88
5.1 电子商务查询	88
5.1.1 产品、类别和评论	88

5.1.2 用户和订单	91
5.2 MongoDB 的查询语言	92
5.2.1 查询条件和选择器	92
5.2.2 查询选择	104
5.3 总结	106
第 6 章 聚合	107
6.1 聚合框架概览	108
6.2 电商聚合例子	109
6.2.1 商品、类别和评价	111
6.2.2 用户和订单	117
6.3 聚合管道操作符	120
6.3.1 \$project	120
6.3.2 \$group	121
6.3.3 \$match、\$sort、\$skip、\$limit	123
6.3.4 \$unwind	123
6.3.5 \$out	124
6.4 重塑文档	124
6.4.1 字符串函数	125
6.4.2 算术运算函数	126
6.4.3 日期函数	126
6.4.4 逻辑函数	127
6.4.5 集合操作符	128
6.4.6 其他函数	129
6.5 理解聚合管道性能	129
6.5.1 聚合管道选项	130
6.5.2 聚合框架的 explain() 函数	130
6.5.3 allowDiskUse 选项	134
6.5.4 聚合光标选项	134
6.6 其他聚合功能	135
6.6.1 .count() 和 .distinct()	135
6.6.2 map-reduce	136
6.7 总结	138
第 7 章 更新、原子操作和删除	140
7.1 文档更新概要	141
7.1.1 通过替换修改	141
7.1.2 通过操作符修改	142
7.1.3 比较两个方法	142
7.1.4 决定：替换与操作符	143

7.2 电商数据模型更新.....	144
7.2.1 商品和目录.....	144
7.2.2 评价.....	148
7.2.3 订单.....	150
7.3 原子文档处理.....	152
7.3.1 订单状态转换.....	153
7.3.2 库存管理.....	155
7.4 核心要点: MongoDB 更新与删除.....	160
7.4.1 更新类型与参数选项.....	160
7.4.2 更新操作符.....	161
7.4.3 findAndModify 命令.....	169
7.4.4 删除.....	169
7.4.5 并发、原子性和隔离.....	170
7.4.6 更新性能注意事项.....	171
7.5 复习更新操作符.....	172
7.6 总结.....	173
第三部分 精通 MongoDB	175
第 8 章 索引与查询优化	177
8.1 索引理论.....	177
8.1.1 精心策划的实验.....	178
8.1.2 核心索引概念.....	181
8.1.3 B-树.....	185
8.2 索引实战.....	186
8.2.1 索引类型.....	186
8.2.2 索引管理.....	189
8.3 查询优化.....	194
8.3.1 找出慢速查询.....	195
8.3.2 检查慢速查询.....	199
8.3.3 查询模式.....	217
8.4 总结.....	219
第 9 章 文本搜索	220
9.1 文本搜索——不仅仅是模式匹配.....	221
9.1.1 文本搜索与模式匹配.....	222
9.1.2 文本搜索与网页搜索.....	223
9.1.3 MongoDB 文本搜索与专业搜索引擎.....	225
9.2 下载曼宁图书类别数据.....	228
9.3 定义文本搜索索引.....	229
9.3.1 文本索引的大小.....	230

9.3.2	分配索引名字并为集合里的所有字段建立索引	231
9.4	基本的文本搜索	232
9.4.1	更复杂的搜索	233
9.4.2	文本搜索分数	235
9.4.3	根据文本搜索分数排序结果	236
9.5	聚合框架文本搜索	237
9.6	文本搜索语言	240
9.6.1	在索引里指定语言	241
9.6.2	在文档里指定语言	242
9.6.3	在搜索中指定语言	243
9.6.4	可用的语言	245
9.7	总结	245
第 10 章	WiredTiger 与可拔插存储	246
10.1	可拔插存储引擎 API	246
10.2	WiredTiger	248
10.2.1	切换到 WiredTiger	248
10.2.2	迁移数据到 WiredTiger	249
10.3	与 MMAPv1 对比	250
10.3.1	配置文件	251
10.3.2	插入脚本与基准测试脚本	252
10.3.3	插入测试结果	255
10.3.4	读性能测试脚本	256
10.3.5	读性能结果	257
10.3.6	测试结论	259
10.4	其他可拔插存储引擎的例子	260
10.5	高级主题	261
10.5.1	可拔插引擎如何工作?	261
10.5.2	数据结构	263
10.5.3	锁	265
10.6	总结	265
第 11 章	复制	267
11.1	复制概览	267
11.1.1	为什么复制很重要	268
11.1.2	复制的使用场景和限制	269
11.2	可复制集	270
11.2.1	安装	270
11.2.2	可复制集群工作原理	277
11.2.3	管理	283

11.3	驱动与复制	291
11.3.1	连接与故障转移	291
11.3.2	写关注点	293
11.3.3	读伸缩	294
11.3.4	标签	296
11.4	总结	298
第 12 章	使用分片集群扩展系统	299
12.1	分片集群概述	300
12.1.1	什么是分片集群	300
12.1.2	什么时候分片?	301
12.2	理解分片集群的组件	302
12.2.1	分片: 存储应用程序数据	303
12.2.2	mongos 路由: 路由操作	303
12.2.3	配置服务器: 存储元数据	303
12.3	在分片集群中分散数据	304
12.3.1	分片集群中的数据分散方式	305
12.3.2	分布式数据库分片	306
12.3.3	集合分片	306
12.4	构建一个例子分片集群	307
12.4.1	启动 mongod 和 mongos 服务器	308
12.4.2	配置集群	310
12.4.3	分片集合	311
12.4.4	写入数据到分片集群	312
12.5	分片集群查询和建立索引	318
12.5.1	查询路由	318
12.5.2	分片集群中建立索引	319
12.5.3	分片集群中的 explain() 工具	320
12.5.4	分片集群中聚合	322
12.6	选择分片键	322
12.6.1	非平衡写入(热点)	323
12.6.2	不可分割的数据块(粗粒度)	324
12.6.3	糟糕的定位(分片键不在查询中)	325
12.6.4	理想的分片键	325
12.6.5	设计折中(email 应用)	326
12.7	生产环境下分片集群	328
12.7.1	配置	328
12.7.2	部署	330
12.7.3	维护	332
12.8	总结	336

第 13 章 部署与管理	337
13.1 硬件与配置	337
13.1.1 集群拓扑	337
13.1.2 部署环境	339
13.1.3 配置	344
13.2 监控与诊断	346
13.2.1 日志	346
13.2.2 诊断命令	347
13.2.3 诊断工具	347
13.2.4 监控服务	349
13.2.5 外部监控应用	349
13.3 备份	350
13.3.1 mongodump 和 mongorestore	350
13.3.2 基于数据文件的备份	351
13.3.3 MMS 备份	352
13.4 安全	352
13.4.1 安全环境	353
13.4.2 网络安全	353
13.4.3 验证	356
13.4.4 可复制集验证	359
13.4.5 分片集群验证	360
13.4.6 企业安全特性	360
13.5 管理任务	360
13.5.1 数据导入和导出	360
13.5.2 压缩和修复	361
13.5.3 升级	363
13.6 性能故障排除	363
13.6.1 工作集	363
13.6.2 性能悬崖	364
13.6.3 查询交互	365
13.6.4 寻求专业帮助	366
13.7 部署检查列表	366
13.8 总结	367
附录 A 安装	368
A.1 安装	368
A.1.1 生产部署	368
A.1.2 32 位和 64 位	369
A.2 Linux 下安装 MongoDB	369

A.2.1	使用预编译二进制文件安装	369
A.2.2	使用包管理器	370
A.3	Mac OS X 下安装 MongoDB	370
A.3.1	预编译二进制版本	370
A.3.2	使用包管理器	371
A.4	Windows 下安装 MongoDB	372
A.5	从源码编译 MongoDB	373
A.6	故障排除	373
A.6.1	错误的架构	373
A.6.2	不存在的数据目录	374
A.6.3	缺少权限	374
A.6.4	未绑定端口	374
A.7	基本配置选项	374
A.8	安装 Ruby	376
A.8.1	Linux 和 Mac OS X	376
A.8.2	Windows	376
附录 B	设计模式	377
B.1	嵌入与引用	377
B.2	一对多	377
B.3	多对多	378
B.4	树	379
B.5	工作队列	382
B.6	动态特性	383
B.7	事务	384
B.8	定位与预计算	385
B.9	反模式	386
B.9.1	粗心索引	386
B.9.2	交错类型	386
B.9.3	单一集合	386
B.9.4	大型、深嵌文档	386
B.9.5	一个用户一个集合	387
B.9.6	不可分片集合	387
附录 C	二进制数据和网格文件系统	388
C.1	简单二进制存储	388
C.1.1	存储缩略图	389
C.1.2	存储 MD5	389
C.2	网格文件	390
C.2.1	Ruby 中的 GridFS	391
C.2.2	使用 mongofiles 操作 GridFS	393

第一部分 入门

Getting started

本书第一部分对MongoDB做了整体介绍，并介绍了实际的开发例子。另外还介绍了JavaScript shell和Ruby驱动，这两个概念会贯穿于本书的所有例子中。

本书主要是面对开发人员，但是如果你是临时使用MongoDB，本书也可以提供很多帮助。虽然我们主要关注MongoDB数据库，但是之前的编程经验会帮助我们理解例子代码。如果之前使用过关系型数据库，那就太棒了！我们会经常对比这两种数据库。

在编写本书的时候，MongoDB 3.0.x版本是最新的版本，但是本书大部分介绍会兼顾之前的MongoDB版本。我们也会强调哪些特性不适用于旧版本。

大部分例子我们会使用JavaScript编写，因为MongoDB JavaScript shell便于实验操作。Ruby是特殊的MongoDB语言之一，我们的例子代码会介绍如何在真实的项目中使用Ruby操作MongoDB数据库。放心，就算大家不是Ruby程序员，我们也可以使用其他语言来开发MongoDB，因为语法都是相似的。

第1章介绍MongoDB的历史、设计目标，以及典型的使用场景。大家也会看到为什么MongoDB可以在与其他NoSQL数据库的竞争中胜出，做到独一无二。

第2章介绍MongoDB shell的用法，让大家熟悉常见的命令工具。我们会介绍MongoDB基本的查询语言，并且通过创建、查询、更新和删除文档来练习实战。本章还会介绍一些高级的shell技巧和MongoDB命令。

第3章主要介绍MongoDB驱动以及BSON数据格式。

这里我们会介绍如何使用Ruby语言来操作数据库，而且会使用Ruby开发一个Demo程序来演示MongoDB的灵活性和强大的查询机制。

要充分掌握本书介绍的知识，请仔细阅读并且亲自尝试每一个例子。如果还没有安装MongoDB，就请参考本书附录A的内容，里面有详细的安装向导。^[1]

^[1]【译注】MongoDB的安装可以参考官方文档 mongodb.org，如果你是其他语言比如 C#、Java 或者 Node 的开发者，也不要担心，官方文档提供了详细的介绍。大家也可以加群 203822816 索取例子代码。

全新Web数据库

A database for the modern Web

本章内容

- MongoDB历史、设计目标以及关键特性
- 简要介绍shell和驱动
- 使用场景和限制
- MongoDB 最新更新

如果你最近几年在开发Web应用，则可能一直使用关系型数据库作为主要的存储方式。如果你熟悉SQL，可能非常喜欢标准化^[1]的数据模型、事务的必要性，以及持久化引擎提供的保证。简单来说就是关系型数据库成熟并且大名鼎鼎。当开发者开始寻找其他替代数据库时，关于新技术的有效性和实用性的疑问不断产生：这些新的数据库要取代传统关系型数据库吗？谁在生产环境中使用它，为什么？迁移到非关系型数据库的动机是什么？其实所有问题的答案归结到一起就只有一个：为什么开发者喜欢MongoDB？

MongoDB是为快速开发互联网Web应用而设计的数据库系统。其数据模型和持久化策略就是为了构建高读/写吞吐量和高自动灾备伸缩性的系统。无论系统需要单个还是多个节点，MongoDB都可以提供高性能。如果你经历过关系型数据库的伸缩困境，那么使用MongoDB就可避免这种困境。但是并非每个人都需要伸缩性操作。如果你需要的就是单台数据库服务器，那么为什么还要使用MongoDB呢？

或许开发者使用MongoDB的最大理由并非是其伸缩策略特性，而是其直观的数据模型。MongoDB在文档里存储数据而不是在行里。什么是文档？下面就是一个例子：

```
{
  "_id": 10,
  "username": 'peter',
  "email": 'pbbakkum@gmail.com'
}
```

这是个非常简单的文档；它存储了用户的一些简单信息字段（听起来很酷）。那这个数据模

^[1]当我们提到标准化的数据模型时，通常是指数据库减少冗余的设计范式。例如，在SQL数据库中我们可以把数据分割为不同的部分，比如Users和Orders表，然后减少用户信息存储的冗余数据，我们称作3NF。

型的优势是什么？设想这种情况：需要为每个用户存储多个email。在关系型数据库里，需要创建单独的email表，然后通过外键关联起来；而MongoDB提供了非常简单的方法来解决这种问题：

```
{
  _id: 10,
  username: 'peter',
  email: [
    'pbbakkum@gmail.com',
    'pbb7c@virginia.edu'
  ]
}
```

如上所示，我们只是创建了一个email数组就解决了问题。作为开发者，我们会发现，这一优点非常有用，在数据经常变化的时候我们只需要存储一个结构化文档即可，无须担心数据模型的变化。

MongoDB的文档格式基于JSON，一种流行的数据存储格式。JSON是JavaScript Object Notation的简称。正如我们看到的，JSON数据结构由键值对组成，也可以内置嵌套。这一点与其他语言中的哈希映射以及字典类型相似。

基于文档的数据模型可以表示丰富的、多层次的数据结构。它经常用来处理无须多表关联的工作。

例如，假设要为电商网站的数据库建模，若使用标准的范式设计数据库，信息可能分割到几十个表中存储，要获得完整的数据库里存储的商品信息，则可能需要关联SQL查询。

相比之下，使用文档模型时，绝大部分单个商品信息都可以存储在单个文档里。打开MongoDB JavaScript shell就可以轻易获取商品的类JSON数据信息。我们也可以查询或者操作它。MongoDB的查询功能是专门用于处理结构化文档操作的，所以用户从关系型数据库到非关系型数据库查询体验基本在同一层次。此外，大部分开发者使用的是面向对象编程语言，他们希望找到更好的数据库用于映射对象。使用MongoDB，语言中定义的对象可以原样持久化保存，减少了对对象映射的复杂性。如果你开发过关系型数据库，这些经验也有助于转换现有技能到新的数据库中。

如果你不熟悉表格数据和对象数据之间的区别，则可能会有很多疑问。不过请放心，学习完第1章你就会对MongoDB的特性以及设计目标有个清晰的认识。我们会先介绍一下MongoDB的发展历史，以及其主要特性。然后会介绍其他NoSQL^[1]解决方案，以及MongoDB是如何解决问题的。最后我们会介绍MongoDB最佳的使用场景，以及它的局限性。

MongoDB在多个方面备受批评，其中有些是公正的，但有些是歪曲的。我们的观点是，它就是开发人员武器库中的一件兵器，和其他数据库一样，你应该知道它的优点和缺点。某些工作需要多表关联和更多的内存管理，而有些工作更适合使用基于文档的数据模型。缺少数据

^[1] 2009年出现的NoSQL一词主要用于描述当时日益流行的非关系型数据库，它们的共同点是使用了SQL之外的查询语言。

架构定义意味着MongoDB更灵活，更适合快速开发模式。我们的目标是告诉大家自己决定MongoDB是否适合自己，以及如何高效地使用它。

1.1 为互联网而生

Built for the Internet

MongoDB的历史不长，但是它诞生于一个雄心勃勃的项目。在2007年，纽约一个叫10gen的创业团队开始工作在一个平台即服务(PaaS)上，由一个应用服务器和一个数据库组成，托管Web应用，根据需要伸缩。与Google App Engine类似，10gen平台的设计目标就是自动处理伸缩和管理硬件与软件基础架构，解放开发者，使得他们可以专注于应用开发。10gen最终发现，开发者并不喜欢放弃对于技术栈的控制，但是用户却喜欢上了10gen的数据库技术。这就导致了10gen团队专注于开发这个数据库产品，最后就形成了MongoDB项目。

10gen公司的名字也已经修改为MongoDB, Inc. 该公司继续支持这个开源项目的开发工作。代码完全公开下载，并且可以免费修改、使用，只要遵守代码开源协议即可。而且鼓励社区提交Bug和补丁。另外，MongoDB核心开发者要么是公司的联合创始人，要么是公司的员工，项目的路线图专注于满足用户社区的需求以及创建兼具关系型数据库和分布式键值存储最佳特性的数据库。因此MongoDB公司的业务模型与其他著名的开源公司不同：支持开源产品的开发并且提供给终端用户订阅服务。

从MongoDB历史中我们了解的最重要的事情就是MongoDB的设计目标就是极简、灵活、作为Web应用栈的一部分。这些使用情况驱动了MongoDB开发过程，并且可以帮助解释它的特性。

1.2 MongoDB 键特性

MongoDB's key features

数据库很大程度上是由其数据模型定义的。本节里，我们会看到文档数据模型，然后会看到MongoDB对此数据模型的高效处理特性。本节里也会介绍其他操作，专注于MongoDB的主从复制以及水平扩展策略。

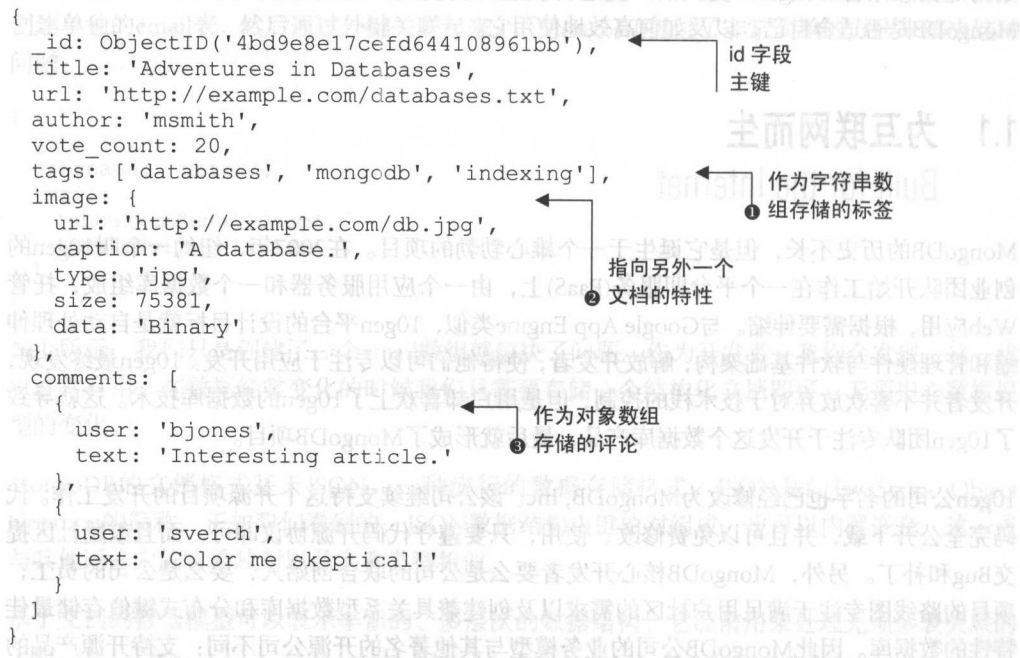
1.2.1 文档数据模型

MongoDB的数据模型是面向文档的。如果对于这里的文档一词不熟悉，则可以通过下面的例子来理解。

JSON文档中除了数值类型以外，其他都需要一对引号。

列表1.1展示了JavaScript版本的JSON文档。这里的引号不是必须的。

列表1.1 表示新闻网入口的文档



这个例子代码展示了新闻网站上篇文章的JSON文档格式（想想Reddit或者Twitter，国内读者若无法访问，则可以类比网易或者今日头条）。正如大家看到的，该文档包含一系列名称和值的集合。这些值可以是简单的数据类型，比如字符串、数字和日期等。当然，这些值也可以是数组，甚至是JSON文档②。后面构造的文档表示了各种不同的数据结构。我们可以看到例子文档包含tags属性①，它把文章的标签存储到数组里。但是最有意思的是comments属性④，它是一个由评论组成的数组。

从内部来讲，MongoDB以二进制JSON格式存储文档数据，或者叫做BSON。BSON有相似的数据结构，但是专门为文档存储设计。当查询MongoDB并返回结果时，这些数据就会转换为易于阅读的数据格式。MongoDB shell使用JavaScript获取JSON格式的文档数据，这也是我们绝大多数例子使用的格式。我们会在后面的章节里深入讨论BSON数据格式。

关系型数据库包含表，MongoDB 拥有集合。换句话说，MySQL在表的行里保存数据，而MongoDB在集合的文档里保存数据，你可以把集合当做一组文档数据。集合是MongoDB中非常重要的概念。集合中的数据存储在磁盘上，而且大部分查询需要指定查询的目标集合。

我们来花点时间来比较MongoDB集合与标准的关系型数据库表示相同数据的差别。图1.1所示为可能的关系结构。因为表本质上是平滑的，表示文档中的文章一对多关系需要多个表。我们可以创建一个posts表存储文章的核心信息。然后可以再创建三个其他的表，每个表包含一个外键字段post_id，关联最初的文章。符合范式设计的数据集保证数据集只在一个地方存储一次。

但是严格的范式设计是需要成本的。值得注意的是，有时候需要一些程序集。要显示刚才关联的文章post，需要执行post和comments表的链接查询。是否需要严格范式设计则最终取决于要建模的数据类型。第4章会深入讨论这个问题。面向文档的数据模型天生就适合表示集中形式的数据，允许我们处理整个数据，从评论到标签，都可以包含在单个数据库对象中。

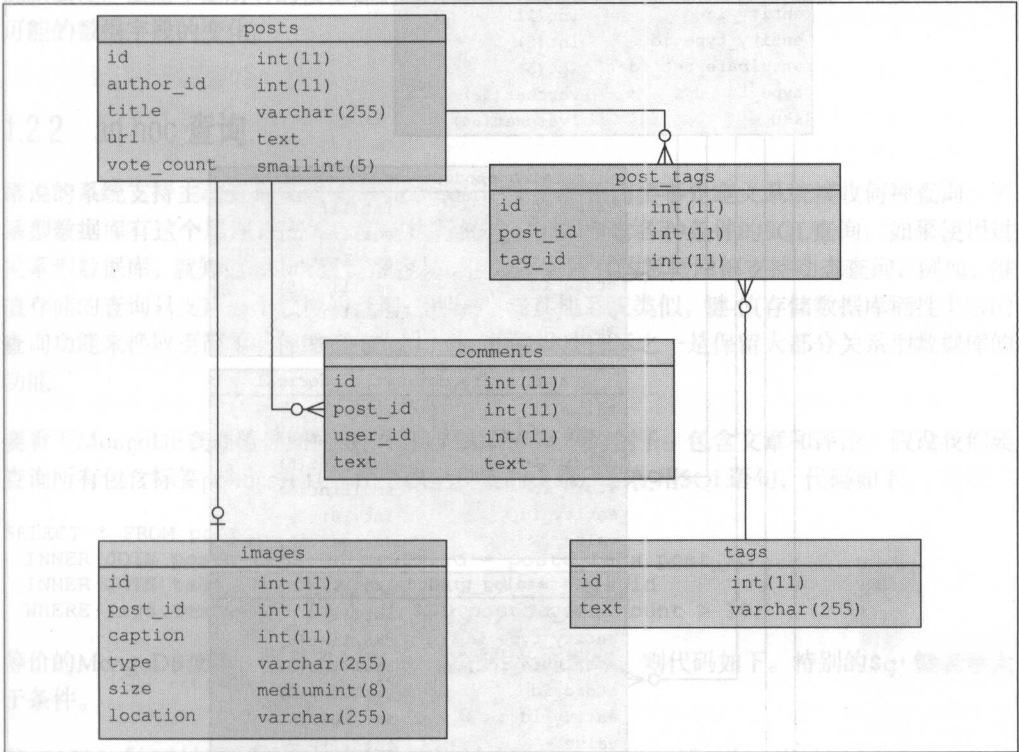


图 1.1 社交新闻网站的实体关联数据模型

十字架线条代表一对一关系，所以一个posts表记录对应一个images记录。三叉线表示一对多关系，所以comments表可以多个评论关联posts表的一条记录。

你可能已经注意到了，除了支持丰富的数据结构外，文档不需要遵守严格的数据定义schema。我们在表里存储行，每个表有严格的schema，指定每个列的数据类型。如果某行需要扩展字段，就必须修改表结构。MongoDB把文档归集到集合中，集合不需要定义任何schema。理论上，每个集合中的文档都可以拥有不同的数据结构；实际上，集合中的文档都是相对一致的。例如，每个posts文章集合中的文档都有标题、标签和评论字段等。

无 schema 模型的优点

不强制定义schema带来了一些好处。首先，应用程序的代码强制数据结构而不是数据库。在频繁修改数据定义的时候这可以加速应用程序开发。

其次，无schema模型允许用户使用真正的变量属性来表示数据。例如，假设你在构建一个电商的商品目录表。由于没有办法知道一个商品包含什么属性，因此应用程序需要处理这些变化。传统的处理方法是在固定schema数据库里使用实体属性值模式^[1]，如图1.2所示。

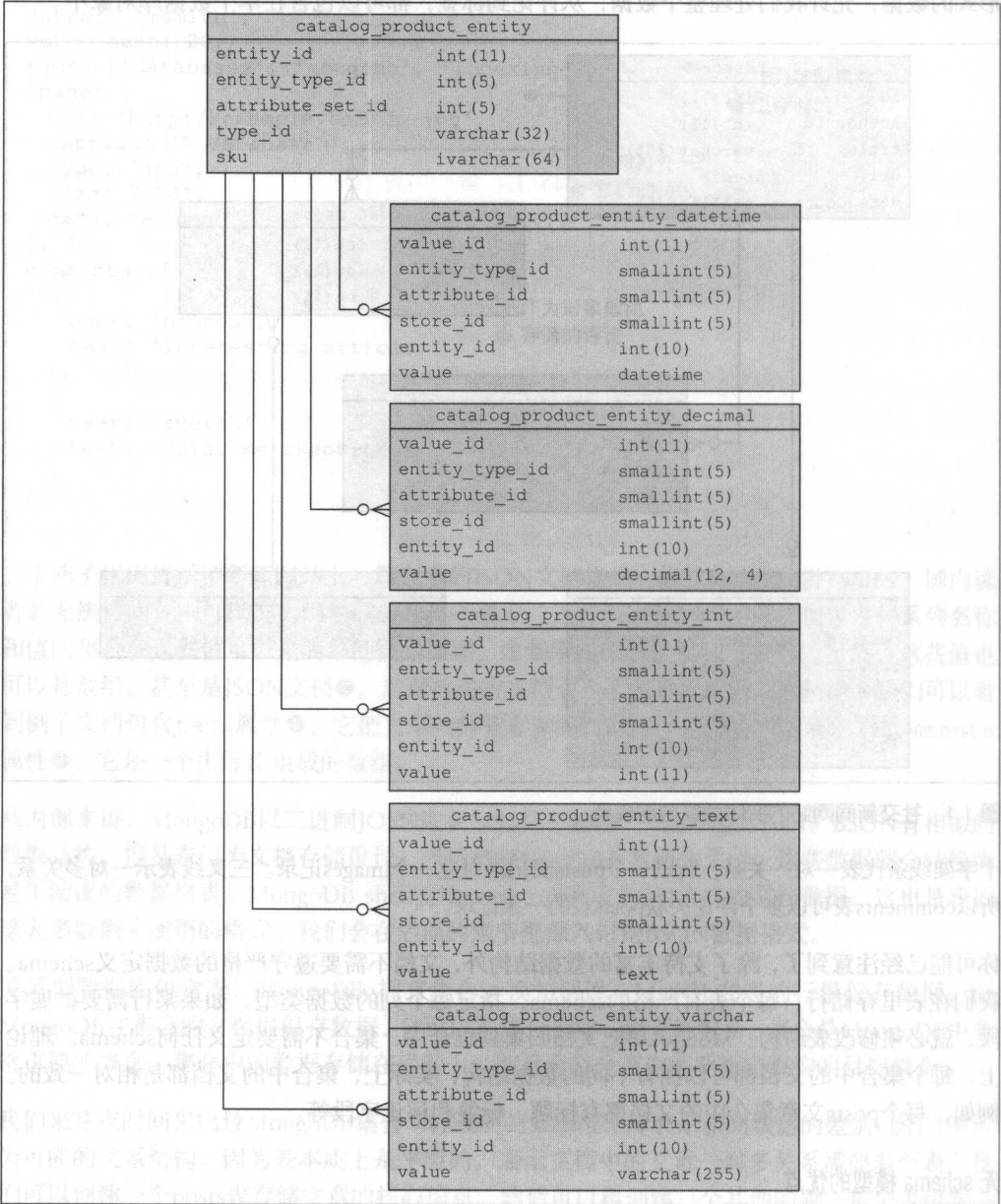


图 1.2 电商应用的部分 schema，这些商品表使用了动态属性创建模式

^[1]更多信息请参考 http://en.wikipedia.org/wiki/Entity-attribute-value_model.

你所看到的只是电商网站的部分数据模型。注意：这些基本都是相似的，除了一个属性值是通过数据类型data type变换的以外。这个数据结构允许管理员来定义额外的产品类型以及属性，但是导致的问题相当复杂。思考一下，如果使用MySQL shell来检查或者更新某个产品模型的数据，则商品SQL关联查询就会相当复杂。而文档建模就不需要关联，而且可以动态添加新属性。虽然不是所有的模型都这么复杂，但是使用MongoDB开发应用就不需要担心未来可能的数据字段的变化。

1.2.2 ad hoc 查询

常说的系统支持主动查询模式（ad hoc queries）是指不需要事先定义系统接收何种查询。关系型数据库有这个属性，它们忠实地执行格式正确的包含各种条件的SQL查询。如果使用过关系型数据库，就知道ad hoc查询很容易。但是不是所有的数据库都支持动态查询，例如，键值存储的查询只支持一个领域的查询：键key。与其他系统类似，键-值存储数据库牺牲丰富的查询功能来换取更简单的伸缩模型。MongoDB的设计目标之一是保留大部分关系型数据库的功能。

要看下MongoDB查询语言如何工作，可先来看个简单的例子：包含文章和评论。假设我们要查询所有包含标签politics并且有10个以上投票的文章。若使用SQL语句，代码如下：

```
SELECT * FROM posts
  INNER JOIN posts_tags ON posts.id = posts_tags.post_id
  INNER JOIN tags ON posts_tags.tag_id == tags.id
 WHERE tags.text = 'politics' AND posts.vote_count > 10;
```

等价的MongoDB查询，使用了document文档作为匹配器，则代码如下。特别的\$gt键表示大于条件。

```
db.posts.find({'tags': 'politics', 'vote_count': {'$gt': 10}});
```

注意：两个查询假设了不同的数据模型。SQL查询依赖于严格的范式模型，posts和tags存储在不同的表里，而MongoDB假设tags保存在每个post文档对象里。但是两个查询演示了任意组合属性的功能，这也是ad hoc查询的功能。

1.2.3 索引

ad hoc查询的一个关键元素就是查找在创建数据库时还不知道的值。随着数据库中增加的文档数据越来越多，查询值的成本变得越来越高。这有时无异于大海捞针。因此，需要一种高效的方式来搜索数据。

理解数据库索引最好的方法就是类比：许多书籍都有索引，包含关键字和页码。假设数据库索引也是提供相似服务的数据结构。假设你有一本秘笈食谱，而你想找出所有和梨子有关的烹饪方法（假设你有许多梨子，又不希望它们坏掉）。耗时的方法就是翻遍图书的所有页面

查找每个方法，逐页查找梨子的做法。而大部分人会选择查看书籍的索引，找出所有包含梨子关键字的食谱列表。

MongoDB中的索引就是使用了B-树（平衡树）数据结构。B-树索引也大量使用于许多关系型数据库中，对于不同的查询做了优化，包括范围扫描和条件子句查询。但是新的引擎已经支持日志结构合并-树(LSM)，可以在MongoDB 3.2版本中使用。

大部分数据库会给每个文档对象一个主键（primary key），一个唯一的数据标识。每个主键会自动索引，这样就可以使用唯一的键来高效地访问每个数据，MongoDB也不例外。但是不是所有数据库都允许我们为单个的行或者文档建立索引。这些叫做辅助索引（secondary indexes）。许多NoSQL数据库，比如HBase，被当做keyvalue数据库，这是因为它们不允许辅助索引（secondary indexes）。通过允许多个辅助索引，MongoDB可以允许用户优化不同的查询。这是MongoDB重要的功能特性。

使用MongoDB，每个集合我们可以创建64个索引。这些索引也可以在其他关系型数据库中找到；升序、降序、复合键、哈希、文本以及地理空间索引^[1]。因为MongoDB和绝大多数关系型数据库RDBMSs使用了相同的索引数据结构，所以管理这些系统的建议都是类似的。你会在下一章里阅读索引的内容，而且必须明白索引对于高效操作数据库至关重要，第8章里会深入介绍这个主题。

1.2.4 复制

MongoDB提供了数据库复制特性，叫做可复制集合（replica set）。可复制集合在多个机器上分布式存储数据，在服务器或者网络出错时，实现数据冗余存储和自动灾备。此外，复制还用于伸缩数据库读操作。如果你在开发读取密集型应用，比如常见的一些网站项目，就可以通过分散读取压力到可复制集群中的服务器来实现。

可复制集合由多态服务器组成。通常，每个服务器有独立的物理机。我们调用这些节点，任意时候，一个节点作为主节点，则其他的作为次节点。与其他数据库中的主从复制类似，可复制集合的主节点可以同时接受读/写操作，但是从服务器只能进行读操作。

真正让可复制集独一无二的是它可以支持自动化灾备：如果主节点失败，则集群中会选择一个从节点，并自动提升为主节点。当之前的主节点回归时，它会继续作为从节点。整个过程如图1.3所述。

复制是MongoDB最有用的特性，我们会在后面的章节里深入讲解。

^[1]地理位置索引允许高效率查询经纬度点信息；会在本书后面进行讨论。

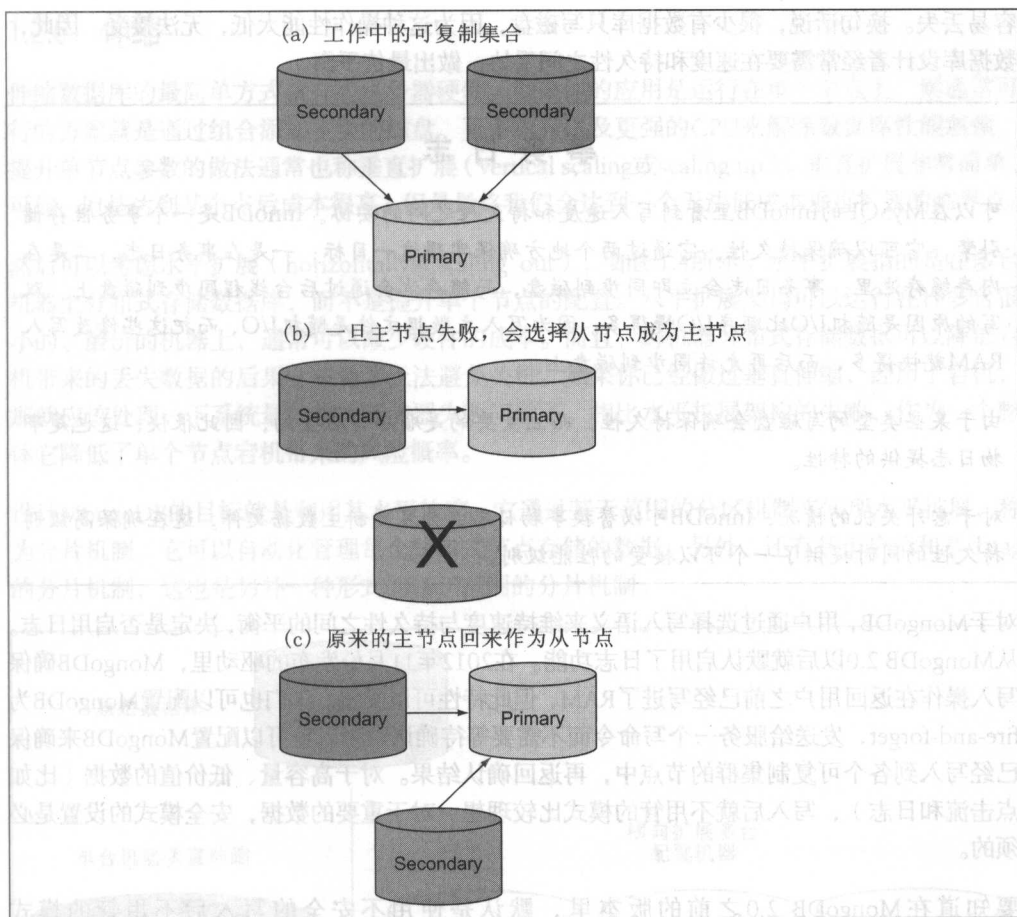


图 1.3 可复制集合的自动化灾备

1.2.5 加速与持久化

要理解MongoDB的持久化方法，首先要思考一些新概念。在数据库系统领域，写入速度和持久性之间存在矛盾的关系。

写入速度（write speed）可以理解为数据库在给定的时间内插入、更新和删除的容量。持久性（durability）指的是这些写操作被永久保存的保证级别。

例如，假如写入100条每条50 KB的记录到数据库中，然后立即切断电源。当重新开机的时候，这些数据还能恢复吗？答案是：取决于数据库系统的配置以及托管硬件。绝大部分数据库默认启动了很好的持久性，所以如果发生这种事情也是安全的。对于一些应用，比如存储日志线，即使可能导致数据丢失，快速写入也很有意义。写入磁盘的速度远远慢于写入内存RAM的速度。特定的数据库，比如Memcached，专门写入RAM，所以它的速度非常快，但是数据

容易丢失。换句话说，很少有数据库只写磁盘，因为这种操作性能太低，无法接受。因此，数据库设计者经常需要在速度和持久性之间妥协，做出最佳平衡。

事务日志

可以在MySQL的InnoDB里看到写入速度和持久性之间的妥协。InnoDB是一个事务性存储引擎，它可以确保持久性。它通过两个地方确保实现这一目标：一是在事务日志，二是在内存缓存池里。事务日志会立即同步到磁盘，而缓存池会通过后台线程同步到磁盘上。双写的原因是随机I/O比顺序I/O慢得多。因为写入主数据文件是随机I/O，而把这些修改写入RAM就快得多，而后再允许同步到磁盘上。

由于某些类型的写磁盘会确保持久性，而且重要的是顺序写磁盘的，因此很快；这也是事物日志提供的特性。

对于意外关机的情况，InnoDB可以替换事物日志，并且更新主数据文件。这在确保高级别持久性的同时提供了一个可以接受的性能级别。

对于MongoDB，用户通过选择写入语义来维持速度与持久性之间的平衡，决定是否启用日志。从MongoDB 2.0以后就默认启用了日志功能。在2012年11月份发布的驱动里，MongoDB确保写入操作在返回用户之前已经写进了RAM，但此特性可以配置。我们也可以配置MongoDB为fire-and-forget，发送给服务一个写命令而不需要等待确认结果。也可以配置MongoDB来确保已经写入到各个可复制集群的节点中，再返回确认结果。对于高容量、低价值的数据（比如点击流和日志），写入后就不用管的模式比较理想。对于重要的数据，安全模式的设置是必须的。

要知道在MongoDB 2.0之前的版本里，默认是使用不安全的写入后不用管的模式fire-and-forget，因为10gen开始开发MongoDB时，只专注于数据层，相信应用层会处理这些错误。但是随着MongoDB使用越来越流行，不仅仅在Web层，对于不想丢失数据的应用来说这样太不安全了。

从MongoDB 2.0开始，日志功能默认是启用的。启用日志功能后，默认100毫秒就会写一次日志文件。如果服务器意外关机，日志会通过重启服务器来确保MongoDB数据文件恢复为一致状态。这是运行MongoDB最安全的方式。

对于写入压力，可以通过关闭日志功能以提高性能。坏处是意外关机之后可能导致数据文件冲突。因此，关闭日志功能，就应该使用主从复制模式，推荐一个从服务器，即使一台机器关机，还有一台机器保证数据的完整性。

设计MongoDB的目标就是让大家可以保持速度与持久性平衡，但是对于重要的数据，我们强烈推荐安全设置。复制和持久化的主题范围很大，我们会在第11章里详细介绍。

1.2.6 伸缩

伸缩数据库的最简单方式就升级服务器硬件。如果你的应用是运行在单个节点上，则通常可行的方案就是通过组合添加更快的磁盘、更多内存以及更强的CPU来解除数据库性能瓶颈。提升单节点参数的做法通常也称垂直扩展（vertical scaling或scaling up）。垂直扩展非常简单、可靠，但是达到某个点后成本很高，但是最终我们会达到一个无法低成本垂直扩展的临界点。

然后可以考虑水平扩展（horizontally或scaling out），如图1.4所示。水平扩展指的是在多台机器上分布式存储数据库，而不是提升单个节点的配置。水平扩展架构可以运行在许多台很小的、廉价的机器上，通常可以减少硬件的成本。而且，跨机器分布式存储数据可以降低宕机带来的丢失数据的后果。服务器无法避免关机。如果你已经做过垂直伸缩，经历了宕机，那就应该处理一下系统最依赖的服务器失败问题了。相比水平扩展架构的失败，作为一个整体它降低了单个节点宕机带来的风险概率。

设计MongoDB的目标就是利用其水平伸缩。它通过基于范围的分片机制来实现水平扩展，称为分片机制，它可以自动化管理每个分布式节点存储的数据。另外，还有基于哈希和基于tag的分片机制，这也是另外一种形式的基于范围的分片机制。

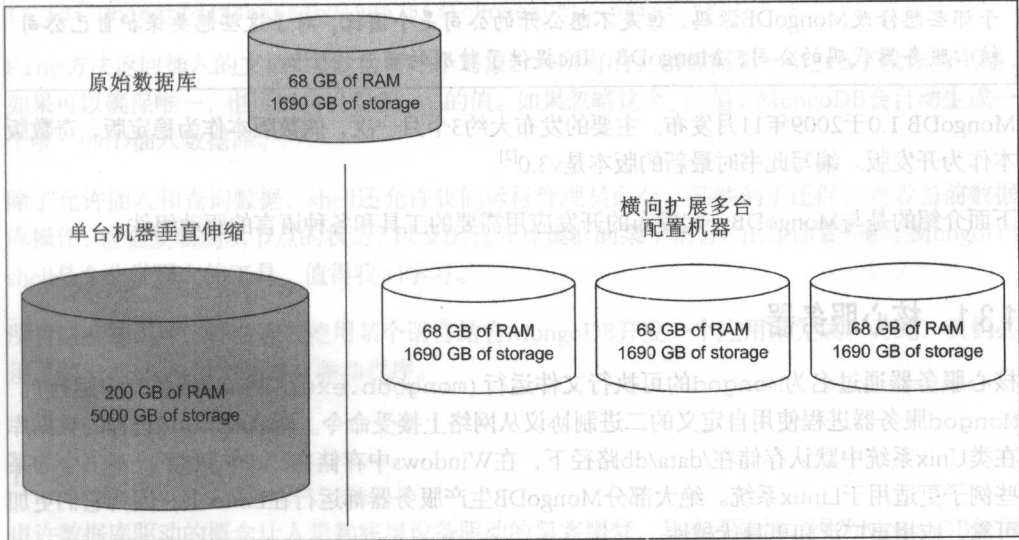


图 1.4 水平与垂直扩展

分片系统处理额外的分片节点，而且它还会处理自动化灾备。每个独立的节点是一个可复制集合，至少由2台机器组成，确保节点失败的时候可以自动恢复。所有这些都意味着没有应用节点必须处理这些逻辑；我们的应用与MongoDB集群通信就好像与单个节点通信一样。第12章会深入讲解分片集群的知识。

我们已经了解了MongoDB最重要的功能特性；第2章里我们会开始学习如何实际开发

MongoDB。现在我们近距离来看看这个数据库。下一节，我们会在它的环境里学习MongoDB，与核心服务器一起发布的工具，以及一些存取数据的方法。

1.3 核心服务和工具

MongoDB's core server and tools

MongoDB使用C++编写，由MongoDB, Inc公司负责开发。这个项目支持主流的操作系统，包括Mac OS X、Windows、Solaris 以及最流行的Linux版本。这些平台的预编译版本可以在<http://mongodb.org>下载。MongoDB是开源的，基于GNU-Affero General Public License (AGPL) 开源协议。源代码可以在GitHub下载，社区也可以贡献力量。但是项目由MongoDB, Inc 核心服务器团队主导，而且他们开发了绝大多数的代码^[1]。

关于 GNU-AGPL 开源协议

GNU-AGPL开源协议存在一些争议。实际上，这个许可协议意味着源代码可以下载，鼓励社区参与。但是GNU-AGPL要求，任何对于源码的修改必须公开发布造福社区。这可能对于那些想修改MongoDB源码，但是不想公开的公司是个困扰。对于这些想要保护自己公司核心服务器代码的公司，MongoDB, Inc提供了特别的商业许可证。

MongoDB 1.0于2009年11月发布。主要的发布大约3个月一次，偶数版本作为稳定版，奇数版本作为开发版。编写此书时最新的版本是v3.0^[2]。

下面介绍的是与MongoDB一起发布的开发应用需要的工具和各种语言的驱动组件。

1.3.1 核心服务器

核心服务器通过名为mongod的可执行文件运行 (mongodb.exe在Windows系统中运行)。Mongod服务器进程使用自定义的二进制协议从网络上接受命令。所有mongod进程的数据库在类Unix系统中默认存储在/data/db路径下，在Windows中存储在c:\data\db下。本书中的某些例子更适用于Linux系统。绝大部分MongoDB生产服务器都运行在Linux上，因为它们更加可靠、应用更广泛和更具优越性。

Mongod可以在几种模式下运行，比如独立模式，或者可复制集群模式。在生产环境中，我们推荐使用可复制群模式MongoDB。通常我们会看到，可复制集群由两台服务器加上一个mongod作为裁判组成。最后，还有一个独立的mongos路由服务器，它用来在分片集群中转发不同的请求到后台服务器。不用担心这些选项，我们会在第11章和第12章中详细介绍这些知识。

^[1]【译者注】翻译此书时最新的版本是3.2.6。

^[2]你应该使用最新的稳定版本，例如 v3.0.6。在附录 A 里有完整的安装指南。

配置mongod进程相对简单，它可以接受命令行参数，也可以接受配置文件文本控制。一些常见的配置可以通过修改mongod侦听的端口和存储数据的目录来实现。要查看这些配置参数，可以运行mongod 帮助文件。

1.3.2 JavaScript shell

MongoDB命令行工具是一个基于JavaScript^[1]的数据库操作和管理工具。Mongo加载命令shell后连接特定的mongod进程，或者默认本地运行。MongoDB shell和MySQL shell类似，最大的不同是它基于JavaScript和SQL脚本。例如，可以选择一个数据库，然后插入一个简单的文档对象到users集合中：

```
> use my_database
> db.users.insert({name: "Kyle"})
```

第一个命令用于选择你要使用的数据库，MySQL用户很熟悉。第二个命令是Javascript表达式，用于插入一个简单的文档。要查看插入的结果，可以使用如下简单的查询：

```
> db.users.find(
{ _id: ObjectId("4ba667b0a90578631c9caea0"), name: "Kyle" }
```

Find方法返回插入的文档数据，带有一个对象ID。所有的文档都需要一个_id字段作为主键。如果可以确保唯一，也可以自己设置_id的值。如果忽略这个_id值，MongoDB会自动生成一个唯一的ID插入数据库。

除了允许插入和查询数据，shell还允许我们运行管理员命令。某些例子还保护查看当前数据库操作，检查复制到从节点的状态，以及配置分片集群的某个集合。正如你看到的，MongoDB shell是个非常强大的工具，值得我们学习。

所有这些知识点，都会通过使用某个语言结合MongoDB开发一个应用来完成。为此，我们必须了解一些MongoDB的语言驱动程序。

1.3.3 数据库驱动

也许数据库驱动的概念让人想起底层设备驱动的黑客噩梦，其实不要怕，因为MongoDB使用起来非常简单。驱动是应用程序用来与MongoDB服务器通信的代码。所有的驱动都保护查询功能、搜索结果、写入数据和运行数据库命令。针对MongoDB驱动开发提供的API都尽量保证所有的语言使用统一的接口。例如，所有的驱动都实现了相似的方法来保存数据到集合中，但是文档的表示类型都是基于自己语言最原始的类型定义。

^[1]如果要学习 Javascript，可以阅读 <http://eloquentjavascript.net> 网站。Javascript 的语法与 C 语言的很像，C++、Java 和 C#都属于 C 语言语系。如果你熟悉其中一种语言，应该很容易理解 Javascript 的例子代码。

在Ruby语言中使用Ruby哈希；在Python中，字典是合适的数据结构类型；在Java中，缺少类似的语言基元类型，通常要使用Map对象表示，或者其他类似的类型。有些开发者喜欢使用ORM框架来管理这些数据的表示工作，但是实际上，MongoDB驱动已经非常强大，这些工作都是多余的^[1]。

语言驱动

在撰写本书时，MongoDB公司官方支持C、C++、C#、Erlang、Java、Node.js、JavaScript、Perl、PHP、Python、Scala、Ruby语言的驱动——这个列表还在增加。如果需要其他语言开发新的驱动，就很有可能已经有社区开发了，虽然并非MongoDB公司官方管理的驱动项目，但是应该也不错。

如果没有你的语言对应的社区支持的驱动，则也可以找到构建新驱动的官方文档规范 <http://mongodb.org>。所有官方的驱动都大量使用在生产环境中，而且遵守Apache许可证，为驱动开发者提供了许多好的参考例子。

从第3章开始，我们将会深入介绍驱动如何工作，以及如何使用它们编写程序。

1.3.4 命令行工具

MongoDB包含了几个命令行工具。

- **mongodump** 和 **mongorestore**——备份和恢复数据库的工具。**mongodump**把数据库数据保存为原生的BSON格式，因此最适用于备份。这个工具的一大优点是适合热备份，而且非常容易使用**mongorestore**命令恢复。
- **mongoexport** 和 **mongoimport**——导入或者导出JSON、CSV、TSV^[2]格式的数据。这在大家需要多种格式的数据时非常有用。**Mongoimport**还可以用来导入大数据集合，只是经常需要在导入之前调整数据模型以便于发挥MongoDB的最大优势。此时最简单的导入数据的方式就是使用自定义脚本。
- **mongosniff**——一个用于查看发送给数据库命令的嗅探工具。通常会把BSON转换为人类可读的shell语句。
- **mongostat**——与**iostat**类似，这个工具用来轮训MongoDB，提供有帮助的状态信息，包括每秒的操作数(增、删、改、查等)，分配虚拟内存的数量，以及服务器的连接数量。
- **mongotop**——与**top**类似，这个工具用来轮训MongoDB，并且显示它在每个集合里花费

^[1] 【译者注】C#语言中，MongoDB不同的客户端驱动也提供了自定义封装的文档类型。

^[2] CSV表示逗号分隔的值(comma-separated value)，意味着使用逗号把数据分割为几块。这是表示表格数据的流行方式，因为列名和行值可以列举在可读的文件中。TSV表示制表符分割的值(tab-separated values)，用制表符Tab来分割数据。

的读取和写入数据的时间总数。

- mongoperf——帮助我们了解MongoDB实例磁盘操作的情况。
- mongooplog——展示MongoDB操作日志里的信息。
- Bsondump——把BSON文件转换为人类可读的格式，包括JSON。

我们会在第2章里详细介绍这些知识点。

1.4 为什么是 MongoDB?

Why MongoDB?

我们已经了解了为什么MongoDB是项目不错的选择的一些原因。这里，我们要更加详细地澄清一下，首先，要考虑一下MongoDB的设计目标。根据MongoDB之父的解释，它被用来设计组合键值对存储和关系数据库的最佳特性。因为简单，所以键值对存储非常快，而且容易伸缩。关系型数据库难以伸缩，至少在水平方向是这样的，但是拥有更加丰富的数据模型和查询语言。MongoDB在两者之间做了妥协，具备了二者的某些有用的功能。它最终的目标就是易于伸缩，存储丰富的数据结构，并且提供复杂的查询语言。

关于使用场景：MongoDB是做Web应用、分析应用的首要数据库。此外，它还比较容易存储无schema数据，也就是弱数据结构的数据。MongoDB也适用于存储无法事先知道数据结构的数据。

这些说法看起来比较虚幻。为了验证这些说法，我们来对比MongoDB和当前使用的不同数据库。接下来你会看到一些MongoDB的专门使用场景，还包括一些生产环境使用的例子。然后，我们会讨论一些使用MongoDB进行实际开发的重要考虑。

1.4.1 MongoDB 与其他数据库对比

数据库的数量非常多，而且对比所有的数据库是不现实的。幸运的是，绝大部分数据库属于某个类别。在表1.1以及接下来的章节里，我们描述了简单而且详细的键值对存储、关系型数据库以及文档数据库，把它们与MongoDB做了对比。

表 1.1 数据库家族

例子	数据模型	伸缩性模型	使用场景
简单的键值存储 Memcached	键值，值是二进制对象	变化的 Memcached 可以跨节点伸缩，把所有可用的 RAM 变为一个存储库	缓存、Web 网站等

	例子	数据模型	伸缩性模型	使用场景
复查键值存储	HBase, Cassandra,	变化的	最终一致性、多节点、分布式高可用和容易灾备	高吞吐量（活动源、消息队列）、缓存、Web 网站等
	Riak KV, Redis, CouchDB	Cassandra 使用的键值结构是列；HBase 和 Redis 存储二进制对象，CouchDB 存储 JSON 文档		
关系型数据库	Oracle 数据库、IBM DB2、Microsoft SQL Server、MySQL、PostgreSQL	表	垂直伸缩。限制支持集群和手动分区	需要事务的系统（银行和金融）或 SQL、规范化数据模型

简单的键值存储

简单的键值存储功能如其名字所示：索引值是基于提供的key键。常见的使用场景就是缓存。例如，假设要缓存App渲染的HTML页面。此时的key可能就是页面的URL，值就是HTML页面本身的数据。注意，对于键值对存储而言，数据值是字节数组。没有强制的schema数据定义，这一点和关系型数据库不同，也没有数据类型概念。这自然也限制了键值存储的操作：可以插入新的值，然后使用key来查询或者删除值。如此简单的系统自然也就非常快速和容易伸缩。

最有名的键值存储就是Memcached，它只在内存里存储数据，是以牺牲持久性来换取速度。它也是分布式的。Memcached节点运行在多个服务器上，也作为单个存储库，去除了跨机器节点保持高速缓存状态的复杂性。

与MongoDB相比，像Memcached这样简单的键值存储允许更快的读/写速度。与MongoDB不同，这些系统不能作为主要的存储数据库。简单的键值存储最好作为辅助手段，或者用作关系型数据库的缓存层，或者作为简单持久性的临时服务，比如工作队列。

复杂的键值存储

可以通过完善简单的键值存储模型来处理复杂的读/写模式或者提供更丰富的数据类型。此时，需要使用复杂的键值存储。其中一个例子就是亚马逊的Dynamo，在一篇非常流行的论文里有介绍，标题是“Dynamo: Amazon’s Highly Available Key-Value Store”（亚马逊高可用键值存储）。Dynamo的设计目标是在网络失败、数据中心故障时可以正常提供强壮的数据库功能，这需要数据可以一直读/写，本质上需要数据自动化复制到各个节点上。如果一个节点失败，系统的用户——此时使用亚马逊购物车——不会经历任何的服务中断。Dynamo提供了一种系统向多个节点写入相同数据时解决不可避免冲突的方法，而且Dynamo易于伸缩。因为它是无主的——所有的节点都一样——这样比较容易理解系统是一个整体，而且可以方便地加入新节点。虽然Dynamo是个亚马逊的数据库系统，但是它的设计思想启发和影响了许多NoSQL

数据库的设计，包括Cassandra、HBase、Riak KV。

通过了解谁开发了这些复杂的键值数据库，以及如何在实际项目中使用，我们可以知道这些系统如何发挥作用。我们来看下Cassandra，它实现了Dynamo的许多伸缩属性，而且受到Google的BigTable启发，提供了面向列的数据模型。Cassandra是个开源数据库，由Facebook构建，其目标是支持站内搜索功能。这个系统水平伸缩，支持索引超过50TB的数据，允许基于用户关键字搜索，属于基于用户ID索引，每条记录都由一组搜索关键字数组和用户ID数组组成，就是为了基于用户进行搜索^[1]。

复杂键值存储由主流的互联网公司开发，比如Amazon、Google、Facebook，用来管理分布式系统中的超大量数据。换句话说，复杂键值存储管理着一个要求大存储与高可用的相对独立的域。因为它们采用了无主架构，系统容易使用额外的节点进行伸缩。它们选择了最终的一致性，意味着读不一定必须反映最新的写。但是为了换取弱一致性，牺牲的是写入时可能出现的节点失败。

对比MongoDB，它提供了更强的一致性、更丰富的数据模型以及辅助索引。后两者特性简单容易；键值存储可以在值里存储任意结构的数据，但是数据库不能查询这些值，除非他们被索引过。也可以使用主键进行查询，或者扫描索引的键，但是如果不使用索引，数据库对于这种查询毫无用处。

关系型数据库

我们已经介绍了很多关系型数据库知识，为了简洁起见，我们只需要讨论RDBMS（关系型数据库管理系统）与MongoDB的异同。流行的关系型数据库包括MySQL、PostgreSQL、Microsoft SQL Server、Oracle Database、IBM DB2等，有些开源，有些不开源。MongoDB和关系型数据库都可以表示丰富的数据模型。而关系型数据库使用固定格式的schema表，MongoDB属于无schema文档。绝大部分关系型数据库支持辅助索引和聚合查询。

从用户角度来看，关系型数据库最大的特性就是支持SQL查询语言。SQL是处理数据强大的工具，但是并非能完美解决所有的工作问题。某些情况下，相比MongoDB，它处理数据时更易于表示和更简单。此外，SQL在各个数据库之间的移植性很强，即使各个数据库支持有点差别。一种思考方式就是，SQL对于数据科学家和分析师来说更容易编写查询语句。MongoDB的查询语言更偏向于开发者，它们在程序里嵌入自己的查询代码。两个模型各有自己的优点和缺点，有时候还取决于个人喜好。

许多关系型数据库支持数据分析（或者数据仓库），而不仅仅是作为数据库。通常数据可以大量导入数据库，然后通过分析来支持商务智能决策问题。这个领域被HP Vertica 或 Teradata Database大公司垄断了，它们都可以提供水平伸缩的SQL数据库。

现在通过在Hadoop存储的数据上运行SQL查询来分析数据的情况越来越多。Apache Hive就是一个广泛使用的工具，可以把SQL查询语句转换为Map-Reduce的工作，这提供了一种伸缩性

^[1] 参见《Cassandra: A Decentralized Structured Storage System》，载于 <http://mng.bz/5321>。

的方式来查询大的数据集。这些查询使用了关系型模型，但是只针对慢速的分析查询，而不是应用程序内置的查询。

文档数据库

很少有数据库标示自己是文档数据库。在编写本书时，对比MongoDB最近的开源数据库是Apache的CouchDB。CouchDB的文档模型与MongoDB的有些相似，虽然数据使用了原始的JSON格式，而MongoDB使用了BSON格式。与MongoDB类似，CouchDB支持辅助索引；不同点在于，CouchDB的索引通过编写mapreduce函数代码实现，相比MySQL 和 MongoDB，其进程不仅仅使用的是声明式语法。在伸缩性方面也不一样，CouchDB不支持跨机器分区；相反，CouchDB节点只是其他节点的完整的复制。

1.4.2 使用场景和部署

坦率地说，你不会只根据数据库特性来选择数据库。我们需要知道真实行业里的成功使用案例。我们来看下MongoDB广泛定义的使用案例，以及一些生产环境下使用的例子^[1]。

Web 应用

MongoDB非常适合做Web应用的主存储数据库。即使是简单的Web应用，也需要使用许多数据模型，比如管理用户、会话、App专有数据、上传以及权限等。这也可以与关系型数据库提供的表格方法很好地兼容，它也可以从MongoDB的集合和文档模型里获取好处。因为文档模型可以表示更丰富的数据结构，需要的集合数量肯定比关系型数据库表的数量要少很多，因为关系型数据库的表需要规范化设计范式。此外，动态查询和索引让我们可以轻易实现绝大多数SQL支持的查询功能。最后，随着Web应用的增长，MongoDB提供了更加清晰的伸缩路径。

MongoDB可以很好地解决高吞吐量的Web网站需求，例如The Business Insider (TBI)案例。它们从2008年1月就开始用MongoDB作为主要的存储库。TBI是一个新闻网站，它每天的吞吐量超过100万独立的页面浏览量。有意思的是，除处理网站主要的内容（文章、评论、用户等）以外，MongoDB还要处理和存储实时的分析数据。这些分析数据被TBI用来生成动态心跳地图，以显示不同新闻的点击量。

敏捷开发

无论你怎么看待敏捷开发，都无法否认快速构建应用系统的热情。许多开发团队，包括Shutterfly和纽约时报，都部分选择了MongoDB，因为它们可以比关系型数据库更快速地开发应用。另外一个显著的原因就是MongoDB没有固定的数据架构定义schema，所以大量节约了花费在提交、沟通和应用schema修改上的时间。

^[1]对于最新的 MongoDB 生产环境部署名单，参见 <http://mng.bz/z2CH>。

此外，花费在将数据关系表示推进到面向对象数据模型中和优化ORM框架生成的SQL语句工作上的时间大大减少了。因此，MongoDB通常适用于实现较短的开发周期的项目，以及敏捷、中型规模的团队。

分析和日志

我们之前说过MongoDB也适用于分析和日志，而且使用MongoDB分析的应用还在增长。通常情况下，一个完善的公司会开始考虑使用MongoDB来做特殊的App分析工作。这些公司包括GitHub、Disqus、Justin.tv、Gilt Groupe等。

MongoDB的相关性分析得益于它的速度和两个特性：针对性原子更新和盖子集合。原子更新允许客户端高效地增加计数器，而且把值推进数组里。盖子集合对于日志非常有用，因为它只存储最近的文档数据。存储日志数据在数据库里与之对应的是文件系统，提供了更简单的组织和更强的查询功能。现在，用户可以使用MongoDB查询来获取日志信息，而不是grep或者自定义搜索工具。

缓存

许多Web应用使用缓存层来帮助快速返回内容数据。允许支持更多对象结构的数据模型（可以把任意文档存储到MongoDB而不需要担心数据结构），结合更快的查询速度，经常允许MongoDB可以作为支持更多查询功能的缓存使用，或者直接与缓存层集成到一起。以The Business Insider网站为例，可以抛弃Memcached，直接从MongoDB处理页面请求。

可变的 Schema

你可以从<https://dev.twitter.com/rest/tools/console>获取一些JSON数据的例子，只要知道如何使用它即可。在获取数据后，保存为sample.Json文件，可以使用如下方式把它导入MongoDB数据库里：

```
$ cat sample.json | mongoimport -c tweets
2015-08-28T11:48:27.584+0300    connected to: localhost
2015-08-28T11:48:27.660+0300    imported 1 document
```

也可以下载一些Twitter流的例子数据，然后直接导入MongoDB集合里。因为流生成了JSON文档，在发给数据库之前不需要修改数据。Mongoimport工具可以直接把数据翻译给BSON。这意味着每个推文微博都会按照自己的格式存储，以一个独立文档存储在集合中。索引和查询内容时，不需要提前声明数据的结构。

你的应用需要调用JSON API时，有这样一个可以自动转换JSON的系统是非常棒的。在存储数据之前不用知道数据结构，而且MongoDB缺少schema约束，因此可以简化我们的数据模型。

1.5 提示和限制

Tips and limitations

对于所有这些好的功能特性，值得记住的是系统权衡和限制。在大家使用MongoDB开发项目之前，我们要先强调一些限制。这些知识都是关于MongoDB如何管理数据，以及如何使用memorymapped文件在磁盘和内存之间移动数据的。

首先，MongoDB通常用于64位系统。32位系统只能寻址4GB内存。这意味着只要你的数据集，包括元数据和存储库达到4GB，MongoDB就无法存储额外的数据。绝大部分生产环境需要的内存比这大，所以64位系统是必须的^[1]。

使用虚拟内存映射的第二个结果是数据内存会根据需要自动分配。这使得在共享环境中运行数据库变得复杂。通常对于数据库服务器，MongoDB最好运行在专门的服务器上。

或许关于MongoDB使用内存映射文件最重要的知识就是它在底层如何处理超过内存大小的数据集。当查询如此大的数据集时，它通常需要通过访问磁盘来获取额外的数据。结果是许多用户报告卓越的MongoDB性能，直到处理的数据超出了内存，而且查询开始变慢。这个问题不仅仅存在于MongoDB中，它也是一个常见的陷阱，值得去留意。

一个关联的问题是MongoDB用来存储集合和文档的数据结构，从数据大小的角度来看并非是最有效的。例如，MongoDB在每个文档里存储key，这意味着对于每个包含“username”字段的文档，都必须使用8个字节来存储该字段的名称。

对于SQL开发者，使用MongoDB常见的痛苦就是，它的查询语言与SQL差别很大，而且有时候确实是事实。MongoDB比绝大多数数据库更针对开发人员——不是分析员。它的哲学是查询一次编写，然后嵌入应用中。正如你将要看到的，MongoDB查询通常由JSON对象组成而不是SQL文本。这使得查询更容易创建和解析，这是个重要的考虑，但是很难为ad-hoc查询去改变。如果你是个分析员，每天要编写查询语句，你会愿意选择使用SQL语句。

最后值得一提的是，虽然MongoDB是最简单的数据库之一，可以作为单个节点运行，但是运行大规模集群还是有维护的成本的。大部分分布式数据库都有类似的问题，而MongoDB更是如此，因为它的集群需要三个配置节点来单独处理分片集群的复制问题。

在一些数据库，例如HBase中，数据存储在每个片中，每个分片数据可以在集群的任意集群上复制。MongoDB不会在所有分片节点上复制数据，而是在每个可复制集群里复制数据。分片和可复制集群是单独的概念，这样有特殊的优势，但是也意味着在配置MongoDB集群时需要单独配置和管理。

我们来快速看下MongoDB里的其他改变。

^[1] 16EB 的内存，几乎可以满足所有的需求和目标。中国 MongoDB 学习交流群 511943641。

1.6 MongoDB 历史

当第一版《MongoDB in Action》出版时，MongoDB 1.8.x是最稳定的版本，2.0.0版本刚刚开始启动。对于本书的第二版，3.0.x是最稳定的版本^[1]。

官方每次重大修改的版本列表如下所示。你最好使用最新的稳定版本，如果是这样，你可以忽略本列表。否则，这个列表可以帮助你决定自己的版本和本书内容的不同。这绝不是一个详尽的列表，因为篇幅限制，我们只列举了每个发布的4~5个项目。

版本 1.8.X

(官方不再支持)

- 分片——分片集群由实验状态修改为产品环境准备状态。
- 可复制集——可复制集状态为产品环境准备。
- 可复制对弃用——可复制集对不再被MongoDB公司支持。
- GEO搜索——引入二维GEO索引（坐标系、2D索引）。

版本 2.0.X

(官方不再支持)

- 默认弃用日志——新版本默认弃用日志功能，日志是阻止数据冲突的重要功能。
- 查询——此版本增加了\$and查询操作符来完善\$or操作。
- 稀疏索引——之前的MongoDB保护每个文档的索引节点，即使文档部包括索引跟踪的字段。稀疏索引只添加包含相关字段的文档节点。这个功能显著降低了索引的大小。某些情况下还可以改善索引的性能，因为小索引可以更有效地使用内存。
- 可复制集优先级——这个版本允许指定可复制集中服务器的优先级，以便于选择新的主服务器。
- 集合级别的压缩和修复——之前的版本只能执行在单个数据库上压缩和修复；这次已扩展到单个的集合中。

版本 2.2.X

(官方不再支持)

- 聚合框架——这个改变使得数据分析和转换更加简单、高效。从某些方面而言，这个工具代替了map/reduce的部分工作；它是基于管道构建，而不是map/reduce模型（难以理解掌握）。

^[1] MongoDB 实际上从 2.6 直接跳到 3.0，忽略了 2.8。参考 <http://www.mongodb.com/blog/post/announcing-mongodb-3.0>，获取更多关于 3.0 的信息。

- **TTL集合**——引入了带有生命周期的集合，允许我们创建与Memcached类似的缓存模型。
- **DB级别锁**——此版本添加了数据库级别的锁来代替全局锁，它通过允许多个操作同时在不同的数据库发生来改善写并发。
- **标签识别分片**——此版本允许节点可以使用ID来标识数据存储的物理位置。这样的应用可以控制数据存储的位置，因此提升效率（只读节点部署在同一个数据中心），减少协作管理的问题（只能在某个国家的服务器上存储该国需要的数据）。

版本 2.4.X

(最老的稳定版本)

- **企业版**——MongoDB的第一个订阅者版本，包括额外的验证模块，可以使用Kerberos验证系统来管理登录数据。免费版包含企业版其他的所有功能。
- **聚合框架性能**——改进聚合框架的性能来支持实时分析。第6章会详细介绍聚合框架。
- **文本搜索**——企业级的搜索方案作为MongoDB的实验特性集成进来。第9章会介绍这个新的搜索功能。
- **增强GEO地理位置索引**——此版本包括支持多边形交叉查询和GeoJSON，以及球星模型的改进，支持椭球模型。
- **V8 JavaScript引擎**——MongoDB以及从Spider Monkey JavaScript引擎切换到Google V8引擎，这改进了多线程操作，并且提升了基于JavaScript的MongoDB map/reduce系统性能。

版本 2.6.X

(稳定发布)

- **\$text查询**——此版本添加了\$text操作符来支持正常查询中的文本搜索。
- **聚合改进**——此版本中聚合有很大的改进。可以在光标上流处理数据，也可以输出数据到集合中。除了其他特性和性能改进，还有许多新增的操作符和管道阶段。
- **为写入改进wire协议**——现在大量写入将会受到更细粒度的应答。批量写入中幸亏有了每次写入的成功或者失败状态，使得写入错误可以通过网络返回给客户端。
- **新更新操作符**——已经为更新操作符添加了\$mul，它可以乘以要更新的值。
- **Sharding改进**——为了更好地处理特定的情况，已经改进了分片集群特性。连续块可以合并，而且重复数据留下来等到数据块迁移完成后自动清理干净。
- **安全改进**——此版本支持集合级别的访问控制，还有用户角色定义。另外还改进了SSL和X509证书支持。
- **查询系统改进**——查询系统的许多部分都被重构过了。这改进了性能和查询的可预测性。
- **企业模块**——MongoDB企业模块改进并扩展了已有的功能，还有审计支持。

版本 3.0.X

(最新的稳定发布版本)

- MMAPv1 存储引擎选择支持集合级别的锁。
- 可复制集选择可以有 50 个成员。
- 支持 WiredTiger 存储引擎；WiredTiger 只有在 MongoDB 3.0 以后的 64 位版本可用。
- WiredTiger 3.0 存储引擎提供了文档级别的锁和压缩功能。
- 可拔插存储引擎 API 允许第三方开发 MongoDB 存储引擎。
- 改进了解释功能。
- SCRAM-SHA-1 验证机制。
- `ensureIndex()` 函数被 `createIndex()` 取代，不应该再使用。

1.7 其他资源

Additional resources

本书的目标是作为教程和权威参考，所以许多语言都是为了介绍主题，并且详细地介绍这些概念。如果需要纯参考，最好的资源是 MongoDB 的用户手册 <http://docs.mongodb.org/manual>。这是一个深入的数据库指南，在需要复习这些概念的时候非常有用，而且我们强烈推荐官方文档。

如果你遇到特别的 MongoDB 问题，可能其他人已经知道了。简单的搜索可能都会返回结果，像博客或者网站 Stack Overflow (<http://stackoverflow.com>) —— 全球最大的面向技术问答的网站。在遇到问题的时候这些都是很大的帮助，但是在用于自己的 MongoDB 之前要详细检查下答案。

你也可以从 MongoDB IRC 聊天组 and 用户论坛获取帮助。MongoDB 公司也提供了咨询服务来帮助企业使用 MongoDB 数据库。许多城市有自己的 MongoDB 用户组，可以通过 <http://meetup.com> 查询。还有一些方式比如接触熟悉 MongoDB 的人来学习如何使用数据库。最后，你也可以在曼宁出版社论坛直接联系我们，《MongoDB in Action》专门的论坛为 <http://manning-sandbox.com/forum.jspa?forumID=677>。在这里可以咨询本书中可能没有介绍的难题，也可以指出漏洞和勘误表。发表问题吧，请不要犹豫！

1.8 总结

Summary

我们已经介绍了许多内容。总结一下，MongoDB 是一个开源的、面向文档的数据库管理系统，

为全新的互联网应用的数据模型和伸缩性而设计，具有动态查询和辅助索引、快速原子更新以及复杂聚合，支持自动化灾备的复制，还有水平伸缩的分片集群等特性。

虽然知识点很多，但是如果你仔细阅读，你可能现在已经急不可耐要为这些功能写代码了。介绍数据库新特性只是其中一个任务，还要在实际中使用它。幸运的是，这就是你在下两章里要学习的知识。首先，大家要熟悉MongoDB JavaScript shell，它可以很方便地与数据库服务引擎交互。其次，在第2章，会开始试验驱动，并尝试构建简单的基于MongoDB的Ruby程序。

第2章 MongoDB JavaScript shell

第3章 MongoDB 驱动

第4章 MongoDB 聚合框架

第5章 MongoDB 索引

第6章 MongoDB 副本集

第7章 MongoDB 分片

第8章 MongoDB 安全

第9章 MongoDB 部署

第10章 MongoDB 性能优化

第11章 MongoDB 高级应用

第12章 MongoDB 进阶应用

第13章 MongoDB 进阶应用

第14章 MongoDB 进阶应用

第15章 MongoDB 进阶应用

第16章 MongoDB 进阶应用

第17章 MongoDB 进阶应用

第18章 MongoDB 进阶应用

第19章 MongoDB 进阶应用

第20章 MongoDB 进阶应用

第21章 MongoDB 进阶应用

第22章 MongoDB 进阶应用

第23章 MongoDB 进阶应用

第24章 MongoDB 进阶应用

第25章 MongoDB 进阶应用

第26章 MongoDB 进阶应用

通过JavaScript shell操作MongoDB

MongoDB through the JavaScript shell

本章内容

- MongoDB shell里使用CRUD操作
- 构建索引和使用explain()
- 掌握基本管理
- 获取帮助

前一章提到了MongoDB运行的经验。如果想学习更多关于实战操作的介绍，本章内容就是。使用MongoDB shell，本章通过一系列实际操作来教会大家基本的数据库知识。你将会学习如何新增、读取、更新和删除(CRUD)文档，在这个过程中熟悉MongoDB查询语言。此外，我们还会初步学习数据库索引，以及如何使用它们来优化查询。然后我们将会学习一些基本的管理命令，建议一些使用MongoDB shell获取帮助的方式。可以把本章作为已经介绍的概念的详细阐述和MongoDB shell操作实战。

MongoDB shell是实验数据库的帮助工具，可以运行ad-hoc查询、管理MongoDB实例。当使用MongoDB开发应用时，我们会使用语言驱动而不是shell，但是shell可以用来做测试和重构这些查询。任意的MongoDB查询都可以在shell里运行。

如果你是MongoDB shell新手，也不要担心，记住它提供了你期望的所有功能；它允许你检查和操作数据，以及管理数据库服务器。MongoDB shell在查询语言方面不同于其他数据库，它没有采用标准的SQL查询语言，我们可以使用Javascript语言和简单的API操作数据库。这意味着你可以在shell里编写脚本与MongoDB数据库交互。如果不熟悉Javascript，没有关系，只需要简单的语法就可以使用shell的功能，本章里所有的例子都会详细解释。MongoDB shell里提供的API与各个语言驱动里提供的接口一样，所以shell里编写的查询代码很容易移植到自己的应用程序代码里。

如果跟着例子进行操作，就会收获很大、受益匪浅，但是必须先安装MongoDB。安装步骤可以在附录A里找到。

2.1 Diving into the 深入 MongoDB shell

Diving into the MongoDB shell

用MongoDB JavaScript shell操作数据库非常简单，而且能够对于文档、集合和数据库查询语言有个实际的感知。下面对MongoDB进行详细介绍。

我们将会从获取和运行shell开始，然后学习用JavaScript如何表示文档，学习如何插入这些文档到MongoDB集合中。要检验这些插入命令，就需要实际查询集合数据、更新操作。最后，通过学习删除数据和集合来完成CRUD操作的练习。

2.1.1 启动 shell

参考附录A，可以很快在本机安装一个可以工作的MongoDB服务和一个运行的mongo实例。通过运行mongo执行文件启动MongoDB shell：

```
mongo
```

如果shell程序启动成功，你的屏幕看起来就会如图2.1所示。shell顶部显示的是运行的MongoDB服务器的版本信息，后面是选择连接的当前数据库信息。

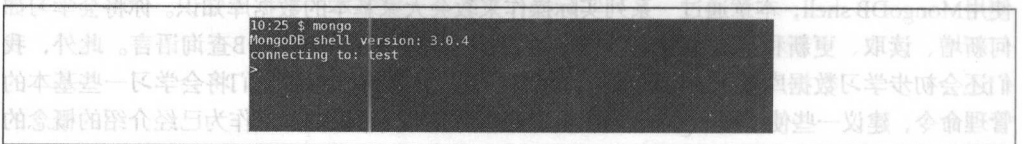


图 2.1 启动 MongoDB JavaScript shell

如果你熟悉JavaScript，就可以开始输入代码，然后探索学习shell。或者，看看如何运行MongoDB数据库的第一个操作命令。

2.1.2 数据库、集合和文档

正如我们知道的，MongoDB把数据存储在文档中，数据可以以JSON (JavaScript Object Notation)格式输出。你可能喜欢在不同的地方存储不同类型的文档，比如users 和 orders。这意味着MongoDB需要一种方式来分类文档，这与关系型数据库RDBMS中的表类似。在MongoDB数据库中，我们称之为集合（collection）。

MongoDB把集合分别存储在不同的数据库中。与传统的SQL数据库不同，MongoDB的数据库只区分集合的命名空间。要查询MongoDB数据库，需要知道存储文档数据的数据库和集合的名字。如果开始没有指定数据库，shell会选择默认的test数据库。为了与后面的练习例子统一，我们切换到tutorial数据库：

```
> use tutorial
switched to db tutorial
```

你会看到一个切换数据库成功的提示信息。

为什么MongoDB有数据库和集合？答案取决于MongoDB如何在磁盘上写数据。数据库中索引的集合都分组在相同的文件中，所以从内存的角度说，这是合理的，保证相关的结合在同一个库里。你可能还希望不同的应用访问同一个集合（多租户），而且保持数据组织性很有帮助，以备未来需求。

创建数据库和集合

你可能好奇，怎样才能切换到tutorial数据库而不需要显示创建它。其实创建数据库不是必须的。只有在第一次插入数据库和集合时才会创建。这个行为符合MongoDB动态操作数据的模式。正如文档的数据结构不需要提前定义一样，单个的数据库和集合也可以在运行时创建。

这样可以简化并加速开发过程。也就是说，你不用担心意外创建数据库或者集合，绝大部分驱动都会阻止这种事情发生。

现在是时候创建第一个文档了。因为使用JavaScript shell，文档使用JSON格式指定，所以简单的文档定义如下所示：

```
{username: "smith"}
```

这个文档包含了一个简单的key和value，用来存储smith的用户名。

2.1.3 插入和查询

要插入文档，就需要选择一个目标集合。我们恰如其分地选择users集合。下面是插入代码：

```
> db.users.insert({username: "smith"})
WriteResult({ "nInserted" : 1 })
```

注意：例子中，我们选择MongoDB shell命令开头带个>符号，这样可以区分输出结果。

你会注意到，在输入这个命令后会出现一些延迟。此刻，既没有在磁盘上创建tutorial数据库，也没有创建users集合。延迟的原因是要为二者分配初始化文件。

如果插入成功，就已经成功保存了第一个文档。在默认的MongoDB数据库配置里，确保已插入这个数据，即使你关闭shell或者重启机器。你可以使用查询来查看新文档数据：

```
> db.users.find()
```

因为数据是users集合的一部分，所以重新打开shell，运行查询，就会显示如下的结果。应答消息如下所示：

```
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

MONGODB 的_id 字段

注意到_id字段已经默认添加到文档里。我们可以把_id字段的值作为文档的主键。每个MongoDB文档都需要一个_id, 而且如果创建文档时没有_id, 就会专门创建一个MongoDB ObjectID添加到文档里。出现在控制台中的ObjectID与代码里列举的不同, 但是在集合中_id值的作用是唯一的, 这是基本要求。可以在文档里插入自己的_id, ObjectID是MongoDB默认的。

下一章会介绍更多的关于ObjectID的知识。我们选择继续添加第二个用户user到集合中:

```
> db.users.insert({username: "jones"})
WriteResult({ "nInserted" : 1 })
```

现在集合中有2个文档了。继续通过运行count来验证下结果:

```
> db.users.count()
```

传递查询条件

现在集合里既然有1个以上文档了, 我们来看下更复杂的查询。和前面一样, 我们先来查询集合里所有的文档:

```
> db.users.find()
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

我们也可以给find方法传递简单的查询选择权。查询选择器是用来匹配集合中文档的。要查询集合中username 为jones的数据, 可以传递简单的条件, 语句如下:

```
> db.users.find({username: "jones"})
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

这个查询条件{username: "jones"}会返回索引名字为jones的文档数据, 它会迭代所有的文档。

注意, 调用find方法若不传递参数, 就等价于传递空条件。也就是说, db.users.find()和db.users.find({})的效果一样。

当然也可以在查询语句中指定多个字段, 这样隐式创建AND语句。例如, 可以使用下面的选择器查询:

```
> db.users.find({
...   _id: ObjectId("552e458158cd52bcb257c324"),
...   username: "smith"
... })
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

程序中的三个点是MongoDB shell自动添加的, 表示这是单行命令。

查询条件会返回等价的文档。条件会使用AND，也就是按并且关系进行查询，索引必须匹配 `_id` 和 `username` 字段。

也可以使用MongoDB的`$and`操作符。之前的查询语句等价修改为

```
> db.users.find({ $and: [
... { _id: ObjectId("552e458158cd52bcb257c324") },
... { username: "smith" }
... ] })
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

使用OR查询文档的语法类似：只需要把`$or`操作符替换一下就可以了。思考下面的语句：

```
> db.users.find({ $or: [
... { username: "smith" },
... { username: "jones" }
... ] })
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

这个查询条件会返回smith和jones的文档，因为我们要找的是名字等于smith或jones的数据。

这个例子和之前的不同，因为它不是简单地插入或查找一个数据文档，而是查询文档本身。在MongoDB里，使用文档来表示命令的做法很普遍，如果你习惯关系型数据库，则可能会感到吃惊。这种做法的一个好处就是，它非常容易在应用中构建自己的查询，因为它们是文档而不是SQL字符串。

我们已经看了基本的创建和读取数据的操作。现在是时候来看一下如何更新数据了。

2.1.4 更新文档

所有的更新至少需要2个参数。第一个指定要更新的文档，第二个定义要如何修改此文档。第一个例子演示了如何修改单个文档，第二个例子演示了如何修改多个文档，甚至如本节的末尾部分一样是集合中的所有文档。但是要记住，默认`update()`只更新一个文档。

通常有两种类型的更新操作，用于不同的属性和使用场景。其中一个类型的更新是在一个文档或者多个文档上修改，另外一个使用新文档取代旧的文档。

从下面的例子我们来看一下简单的文档：

```
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

更新操作符

第一种类型的更新需要传递一个文档参数，还有一些操作符作为第二个参数。本节里，我们先来看下如何使用`$set`操作符，它可以为单个字段设置特定的值。

假设用户smith要增加自己的居住地国家，我们可以使用如下命令更新：


```
> db.users.update({username: "smith"}, {$set: {country: "Canada"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

这个命令会告诉MongoDB，找到一个用户名为smith的文档，然后把country属性设置为Canada。我们可以在服务器返回的消息里看到更新结果。如果查询被修改的数据，就可以看到该文档已经被更新：

```
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith",
  "country" : "Canada" }
```

替换更新

另外一个更新文档的方式就是替换文档，而不是更新某个字段。当使用\$set操作符的时候这一点容易混淆。思考下面不同的更新命令：

```
> db.users.update({username: "smith"}, {country: "Canada"})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

这个例子里，文档被替换为只包含country字段的文档。username字段被删除，因为它只是用来匹配文档，第二个参数用来更新替换。当我们在进行这种更新时应该多加注意。新文档的查询如下所示：

```
> db.users.find({country: "Canada"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "country" : "Canada" }
```

_id相同，但是数据已经被替换为新的文档了。当确定是新增或者修改数据而不是替换整个文档时，就使用\$set操作符。

把用户名username重新添加到记录里：

```
> db.users.update({country: "Canada"}, {$set: {username: "smith"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.users.find({country: "Canada"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "country" : "Canada",
  "username" : "smith" }
```

如果以后不想要country字段了，使用\$unset操作符删除即可：

```
> db.users.update({username: "smith"}, {$unset: {country: 1}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

更新复杂数据

我们来丰富一下以上例子。正如在第1章里看到的，我们使用文档来标识数据。文档可以包含复杂的数据。假设除了保存配置信息，用户还可以保存自己喜欢的东西。符合要求的文档格式可能看起来如下：

```
{
  username: "smith",
  favorites: {
    cities: ["Chicago", "Cheyenne"],
    movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
  }
}
```

favorites键指向了一个包含2个键的新对象，它包含喜欢的城市和电影。假设你已经弄懂了，就可以把之前smith的文档修改为这个格式了。此时应该想起了\$set操作符了：

```
> db.users.update( {username: "smith"},
... {
...   $set: {
...     favorites: {
...       cities: ["Chicago", "Cheyenne"],
...       movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
...     }
...   }
... })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

请注意，间距缩进不是必须的，但是这样可以避免错误，文档的可读性更强。

下面我们来用相似的方式修改jones数据。此时，我们只能添加一些喜欢的电影：

```
> db.users.update( {username: "jones"},
... {
...   $set: {
...     favorites: {
...       movies: ["Casablanca", "Rocky"]
...     }
...   }
... })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

如果输入错误，则可以使用向上的方向键回到前一条语句。

选择查询users集合来确保两个更新成功：

```
> > db.users.find().pretty()
{
  "_id" : ObjectId("552e458158cd52bcb257c324"),
  "username" : "smith",
  "favorites" : {
    "cities" : [
      "Chicago",
      "Cheyenne"
    ],
    "movies" : [
      "Casablanca",
      "For a Few Dollars More",
      "The Sting"
    ]
  }
}
```

```

}
{
  "_id" : ObjectId("552e542a58cd52bcb257c325"),
  "username" : "jones",
  "favorites" : {
    "movies" : [
      "Casablanca",
      "Rocky"
    ]
  }
}

```

严格来说, `find()` 命令文档返回一个 `cursor` 光标。因此, 要访问文档, 就需要迭代光标。`find()` 命令自动返回20个文档——如果可用——就会迭代光标20次。

有了这些例子文档, 我们可以开始体验MongoDB查询语言的强大之处了。特别是, 查询引擎可以深入内嵌对象的内部, 以及根据数据元素进行匹配, 这些都是非常有用的功能。注意, 我们是通过给 `find` 操作附加 `pretty` 操作来获取从服务器端返回的良好格式的结构。严格来说, `pretty()` 实际上就是 `cursor.pretty()`, 它可以配置光标以容易阅读的方式来显示结果。

我们可以在本次查询里看到这两个概念的演示例子, 找出所有喜欢电影Casablanca (卡萨布兰卡) 的用户:

```
> db.users.find({"favorites.movies": "Casablanca"})
```

`favorites` 和 `movies` 之间的原点会告诉查询引擎来搜索一个键名为 `favorites` 的对象, 内部键名为 `movies` 作为新的匹配条件。因此, 这个查询返回2个用户文档, 如果数组中的元素匹配了最初查询, 数组上的查询将会匹配。

假设你知道任意喜欢Casablanca的用户也会喜欢电影The Maltese Falcon, 而且想要用更细数据库来反映这个事实, 那么就要看更复杂的查询。如何在MongoDB更新命令里实现呢?

高级更新

你可能想到再次使用 `$set` 操作符, 但是这样做需要重新编写并发送整个电影数组。因为我们想要做的就是给列表添加元素, 最好还是使用 `$push` 或者 `$addToSet`。这2个命令都是往数组中添加数据, 但是第二个是唯一的, 阻止了重复的数据。

这就是你要的更新语句:

```
> db.users.update( {"favorites.movies": "Casablanca"},
...   { $addToSet: { "favorites.movies": "The Maltese Falcon" } },
...   false,
...   true )
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })

```

这个代码容易理解。第一个参数是查询条件, 匹配电影列表中包含Casablanca的用户。第二个参数使用 `$addToSet` 添加The Maltese Falcon到列表中。

第三个参数false, 控制是否允许upsert。这个命令告诉更新操作, 当一个文档不存在的时候是否插入它, 这取决于更新操作是操作符更新还是替换更新。

第四个参数true, 表示是否是多个更新。默认情况下, MongoDB更新只针对第一个匹配文档。如果想更新所有匹配的文档, 就必须显示指定这个参数。如果你想对smith和jones都更新数据, 那么这个参数是必须的。

我们将会详细介绍更新, 但是在继续学习之前先练习一下这些例子。

2.1.5 删除数据

现在我们已经学习了通过MongoDB Shell 创建、读取和更新数据的基本操作。最后我们来学习下最简单的操作: 删除数据。

如果没有参数, 删除操作将会清空集合里的所有文档。如果要清空foo集合里的所有文档内存, 可以输入以下命令:

```
> db.foo.remove()
```

通常我们只需要删除集合中某个文档, 因此, 我们要传递查询选择器给remove()方法。如果要删除所有喜欢Cheyenne城市的用户, 则可以这样编写简单的表达式:

```
> db.users.remove({"favorites.cities": "Cheyenne"})
WriteResult({"nRemoved" : 1 })
```

注意: remove()操作不会删除集合, 它只会删除集合中的某个文档。我们可以把它和SQL中的DELETE命令进行类比。

如果要删除集合及其附带的索引数据, 可以使用drop()方法:

```
> db.users.drop()
```

创建、读取、更新和删除都是任意数据库的基本操作。如果你已经用过, 那么可以直接在MongoDB里进行CRUD的操作练习。下一节里, 我们将会学习如何通过辅助索引来增强查询、修改和删除。

2.1.6 shell 的其他特性

你可能已经注意到了, shell可以很方便地做很多与MongoDB有关的工作。我们可以使用上下方向键快速切换之前的命令, 使特定的输入自动完成, 比如集合名字。自动完成特性使用tab键来自动完成或者列举完成的可能情况^[1]。

^[1]完整的快捷键列表, 请参考 <http://docs.mongodb.org/v3.0/reference/program/mongo/#mongo-keyboard-shortcuts>。

我们也可以输入help来获取更多关于shell的帮助信息:

```
> help
```

许多函数也可以打印帮助信息,用来提示如何使用函数。试一下:

```
> db.help()
DB methods:
  db.adminCommand(nameOrDocument) - switches to 'admin' db, and runs
  command [ just calls db.runCommand(...) ]
  db.auth(username, password)
  db.cloneDatabase(fromhost)
  db.commandHelp(name) returns the help for the command
  db.copyDatabase(fromdb, todb, fromhost)
:
```

查询的帮助通过一个不同的函数explain来提供。我们会在后面详细讲解。在启动MongoDB shell时还有许多参数选项可用。要显示这个列表,可以在启动MongoDB shell时指定help标签:

```
$ mongo --help
```

你不需要担心怎样使用这些功能特性,我们还没有详细介绍shell呢!但是它确实可以提供我们需要的帮助信息。

2.2 使用索引创建和查询

Creating and querying with indexes

通过创建索引来改善查询性能是常见的做法。幸运的是, MongoDB的索引可以方便地从shell创建。如果你还不了解数据库索引,则本节可以帮你弄清楚这些概念;如果你曾经使用过索引,就会发现创建索引非常简单。使用explain()方法监控查询。

2.2.1 创建大集合

索引只有在集合包含许多文档的时候才有意义。我们先往numbers集合里添加2000个文档。因为MongoDB shell也是个JavaScript解析器,代码实现起来非常简单:

```
> for(i = 0; i < 20000; i++) {
  db.numbers.save({num: i});
}
WriteResult({"nInserted": 1 })
```

因为文档很多,所以如果花费了几秒钟也不要惊讶。一旦返回,我们可以运行一些查询语句来检验所有显示的文档:

```
> db.numbers.count()
20000
```



```
> db.numbers.find()
{ "_id": ObjectId("4bfbf132dbalaa7c30ac830a"), "num": 0 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac830b"), "num": 1 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac830c"), "num": 2 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac830d"), "num": 3 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac830e"), "num": 4 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac830f"), "num": 5 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac8310"), "num": 6 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac8311"), "num": 7 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac8312"), "num": 8 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac8313"), "num": 9 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac8314"), "num": 10 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac8315"), "num": 11 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac8316"), "num": 12 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac8317"), "num": 13 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac8318"), "num": 14 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac8319"), "num": 15 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac831a"), "num": 16 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac831b"), "num": 17 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac831c"), "num": 18 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac831d"), "num": 19 }
Type "it" for more
```

count() 命令显示已经插入了20000个文档。下一个查询显示了前20条结果 (shell里的结果可能不太一样)。

我们可以使用it命令来显示其余的结果:

```
> it
{ "_id": ObjectId("4bfbf132dbalaa7c30ac831e"), "num": 20 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac831f"), "num": 21 }
{ "_id": ObjectId("4bfbf132dbalaa7c30ac8320"), "num": 22 }
:
```

it命令告诉shell返回下一个结果集^[1]。

既然已经有了大量的测试数据，我们来尝试一些新的查询。假设你已经熟悉了MongoDB的查询引擎，则匹配文档属性num的查询语句如下：

```
> db.numbers.find({num: 500})
{ "_id" : ObjectId("4bfbf132dbalaa7c30ac84fe"), "num" : 500 }
```

范围查询

更有意思的是，你可以使用\$gt和\$lt运算符来进行范围查询。这两个符号表示大于和小于。假如你要查询所有num值大于19995的所有文档，则语句如下：

```
> db.numbers.find( {num: { "$gt": 19995 }} )
{ "_id" : ObjectId("552e660b58cd52bcb2581142"), "num" : 19996 }
{ "_id" : ObjectId("552e660b58cd52bcb2581143"), "num" : 19997 }
```

^[1] 你可能好奇背后发生的事情。所有的查询都创建一个游标，以便于在结果集上迭代。这是使用shell时候看不到的工作，所以此时没有必要讨论太详细。如果你等不及学习光标及其特性，那可以直接阅读第3章和第4章的内容。

```
{ "_id" : ObjectId("552e660b58cd52bcb2581144"), "num" : 19998 }
{ "_id" : ObjectId("552e660b58cd52bcb2581145"), "num" : 19999 }
```

可以结合使用两个运算符，设置上下限：

```
> db.numbers.find( {num: {"$gt": 20, "$lt": 25}} )
{ "_id" : ObjectId("552e660558cd52bcb257c33b"), "num" : 21 }
{ "_id" : ObjectId("552e660558cd52bcb257c33c"), "num" : 22 }
{ "_id" : ObjectId("552e660558cd52bcb257c33d"), "num" : 23 }
{ "_id" : ObjectId("552e660558cd52bcb257c33e"), "num" : 24 }
```

可以使用简单的JSON文档查看，你也可以用与SQL一样的方式来指定范围。`$gt`和`$lt`是MongoDB查询语言里两个主要的运算符。`$gte`表示大于等于，`$lte`表示小于等于，而`$ne`表示不等于。我们会在后续的章节里看到更多的运算符。

当然，除非这种查询非常高效，否则也没有价值。下一节，我们会思考如何通过MongoDB的索引功能来提升查询效率。

2.2.2 索引和 explain()

如果你使用过关系型数据库，则可能对于SQL的EXPLAIN非常熟悉了，它是一个调试和优化查询的好工具。当所有数据库接收到查询后，它必须弄清楚如何执行查询；这就称为查询计划。EXPLAIN描述了查询路径并且允许开发者通过确定查询使用的索引来诊断慢的查询语句。查询通常可以有多种方式执行，但有时候并非我们期望的方式^[1]。MongoDB 有自己的EXPLAIN版本，它可以提供相同的功能。

要了解其工作原理，我们来分析一下刚才已经使用的查询语句。在系统里运行下面的命令：

```
> db.numbers.find({num: {"$gt": 19995}}).explain("executionStats")
```

结果应该与列表2.1里显示的类似。`"executionStats"`关键字是MongoDB 3.0新增的，请求不同的模式并输出更详细的信息。

列表2.1 未使用索引的典型的`explain("executionStats")`输出结果

```
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "tutorial.numbers",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "num" : {
        "$gt" : 19995
      }
    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "num" : {

```

^[1]【译者注】SQL Server 有执行计划，可视化工具，可以优化分析 SQL 语句的性能。

```

        "$gt" : 19995
    },
    "direction" : "forward"
},
"rejectedPlans" : [ ]
},
"executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 4,
    "executionTimeMillis" : 8,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 20000,
    "executionStages" : {
        "stage" : "COLLSCAN",
        "filter" : {
            "num" : {
                "$gt" : 19995
            }
        },
        "nReturned" : 4,
        "executionTimeMillisEstimate" : 0,
        "works" : 20002,
        "advanced" : 4,
        "needTime" : 19997,
        "needFetch" : 0,
        "saveState" : 156,
        "restoreState" : 156,
        "isEOF" : 1,
        "invalidates" : 0,
        "direction" : "forward",
        "docsExamined" : 20000
    }
},
"serverInfo" : {
    "host" : "rMacBook.local",
    "port" : 27017,
    "version" : "3.0.6",
    "gitVersion" : "nogitversion"
},
"ok" : 1
}

```

看到explain()的输出结果^[1]，你可能感到惊讶，居然查询引擎会扫描整个集合（即20000个文档）(docsExamined)，而只返回了4个结果。totalKeysExamined字段显示了整个扫描的索引数量，它的值是0。

扫描文档的数量和低效查询的数量居然差距这么大。现实的情况是集合和文档可能更大，实际查询消耗的时间可能会更多，超过8 ms（不同的机器可能不太一样）。

集合需要的就是一个索引。我们可以使用createIndex()方法为num 键创建索引。可以输入以

^[1] 这些例子里我们插入“hostname”作为机器主机名。在你的机器上可能是 localhost，或者机器名或者名字加上.local。不要担心与我们的结果不同，这和平台的版本还有 MongoDB 版本有关系。

下命令:

```
> db.numbers.createIndex({num: 1})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

在MongoDB 3以后的版本, `createIndex()` 方法取代了 `ensureIndex()` 方法。如果你使用的是旧版本的MongoDB, 则可以继续使用 `ensureIndex()` 而不用 `createIndex()`。在MongoDB 3中, `ensureIndex()` 依然可用, 它是 `createIndex()` 的别称。

对于其他的MongoDB操作, 比如查询和更新, 要传递文档给 `createIndex()` 方法来确定索引的键。此时 `{num: 1}` 文档表示应该为 `numbers` 集合中的所有文档的 `num` 键建立升序索引。

我们可以通过 `getIndexes()` 方法来检验索引是否创建成功:

```
> db.numbers.getIndexes()
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "tutorial.numbers"
  },
  {
    "v" : 1,
    "key" : {
      "num" : 1
    },
    "name" : "num_1",
    "ns" : "tutorial.numbers"
  }
]
```

集合现在有2个索引。第一个是标准的 `_id` 索引, 自动为每个集合构建的; 第二个是我们自己创建的 `num` 索引。这些索引的名字分别叫 `_id_` 和 `num_1`。如果没有设置名字, MongoDB会自动创建一个有意义的名字给它们。

如果使用 `explain()` 进行查询, 可以看到应答时间有很大改变。具体如列表2.2所示。

列表2.2 `explain()`输出的索引查询

```
> db.numbers.find({num: {"$gt": 19995 }}).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "tutorial.numbers",
    "indexFilterSet" : false,
    "parsedQuery" : {
```

```

    "num" : {
      "$gt" : 19995
    },
  },
  "winningPlan" : {
    "stage" : "FETCH",
    "inputStage" : {
      "stage" : "IXSCAN",
      "keyPattern" : {
        "num" : 1
      },
      "indexName" : "num_1",
      "isMultiKey" : false,
      "direction" : "forward",
      "indexBounds" : {
        "num" : [
          "(19995.0, inf.0]"
        ]
      }
    },
    "rejectedPlans" : [ ]
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 4,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 4,
    "totalDocsExamined" : 4,
    "executionStages" : {
      "stage" : "FETCH",
      "nReturned" : 4,
      "executionTimeMillisEstimate" : 0,
      "works" : 5,
      "advanced" : 4,
      "needTime" : 0,
      "needFetch" : 0,
      "saveState" : 0,
      "restoreState" : 0,
      "isEOF" : 1,
      "invalidates" : 0,
      "docsExamined" : 4,
      "alreadyHasObj" : 0,
      "inputStage" : {
        "stage" : "IXSCAN",
        "nReturned" : 4,
        "executionTimeMillisEstimate" : 0,
        "works" : 4,
        "advanced" : 4,
        "needTime" : 0,
        "needFetch" : 0,
        "saveState" : 0,
        "restoreState" : 0,
        "isEOF" : 1,
      }
    }
  }
}

```

使用 num_1 索引

只扫描 4 个文档

更快

返回 4 个文档


```

    "invalidates" : 0,
    "keyPattern" : {
      "num" : 1
    },
    "indexName" : "num_1",
    "isMultiKey" : false,
    "direction" : "forward",
    "indexBounds" : {
      "num" : [
        "(19995.0, inf.0]"
      ]
    },
    "keysExamined" : 4,
    "dupsTested" : 0,
    "dupsDropped" : 0,
    "seenInvalidated" : 0,
    "matchTested" : 0
  }
},
"serverInfo" : {
  "host" : "rMacBook.local",
  "port" : 27017,
  "version" : "3.0.6",
  "gitVersion" : "nogitversion"
},
"ok" : 1
}

```

使用 num_1 索引

现在查询使用了 num 键的 num_1 索引，它只扫描了与查询有关的 4 个文档。整个查询消耗的总时间从 8 ms 降低到 0 ms!

索引并非没有成本，它们会占用空间，而且会让插入成本稍微提升，对于查询优化来说这是必备的工具。如果对这个例子理解有些疑问，可以阅读本书第 8 章，第 8 章专门深入讲解了索引和查询优化机制。接下来我们要学习关于 MongoDB 实例的管理命令。我们也会学习如何从 shell 里获取帮助，这有助于我们掌握各种不同的 shell 命令。

2.3 基本管理

Basic administration

本章承诺要介绍通过 JavaScript shell 管理 MongoDB。我们已经学习了基本的数据操作和索引。这里，我们将会介绍获取 mongod 进程信息的方法。例如，你可能想知道不同的集合占用的存储空间，或者集合中定义了多少索引。命令返回的信息可以帮助我们来诊断性能问题并且跟踪分析数据。

我们将了解一下 MongoDB 的命令接口。最特殊的就是，在 MongoDB 上执行的是非 CRUD 操作，从服务器状态检查到数据文件完整性验证，都使用数据库命令来实现。我们将会解释 MongoDB 上下文里的命令，以及展示它使用起来有多么简单。最后，最好要知道去哪里获取

帮助。我们后面将会告诉大家在shell里如何获取深入学习MongoDB的更多帮助。

2.3.1 获取数据库信息

通常，我们想知道安装MongoDB的服务器上存在什么集合和数据库。

幸运的是，MongoDB shell提供了许多命令，包括一些语法糖，用来获取系统的信息。

show dbs打印系统中所有的数据库列表信息：

```
> show dbs
admin (empty)
local 0.078GB
tutorial 0.078GB
```

show collections展示了当前数据库里所有的集合^[1]。如果选择的还是tutorial数据库，就会看到这个集合。后面我们还会继续使用这个集合。

```
> show collections
numbers
system.indexes
users
```

有个可能你不认识的集合system.indexes。这个是每个数据库都存在的特殊集合。system.indexes中的每个入口都定义了一个数据库索引，我们可以使用getIndexes()方法来查看，正如之前我们看到的一样。但是MongoDB 3.0放弃了对于system.indexes集合的访问，我们可以使用createIndexes或者listIndexes替换。getIndexes() JavaScript方法可以被db.runCommand({"listIndexes": "numbers"}) shell命令取代。

对于低级别数据库和集合分析，stats()方法非常有用。当我们在数据库对象上运行此方法时，就会获取下面的结果：

```
> db.stats()
{
  "db" : "tutorial",
  "collections" : 4,
  "objects" : 20010,
  "avgObjSize" : 48.0223888055972,
  "dataSize" : 960928,
  "storageSize" : 2818048,
  "numExtents" : 8,
  "indexes" : 3,
  "indexSize" : 1177344,
  "fileSize" : 67108864,
  "nsSizeMB" : 16,
  "extentFreeList" : {
    "num" : 0,
```

^[1]这个结果里显示的有些信息只有在复杂的调试或者优化时才会有用。但是我们至少也可以了解某个集合以及索引占用的空间。

```

    "totalSize" : 0
  },
  "dataFileVersion" : {
    "major" : 4,
    "minor" : 5
  },
  "ok" : 1
}

```

我们也可以单个集合上执行stats()：

```

> db.numbers.stats()
{
  "ns" : "tutorial.numbers",
  "count" : 20000,
  "size" : 960064,
  "avgObjSize" : 48,
  "storageSize" : 2793472,
  "numExtents" : 5,
  "nindexes" : 2,
  "lastExtentSize" : 2097152,
  "paddingFactor" : 1,
  "paddingFactorNote" : "paddingFactor is unused and unmaintained in 3.0. It remains hard coded to 1.0 for compatibility only.",
  "systemFlags" : 1,
  "userFlags" : 1,
  "totalIndexSize" : 1169168,
  "indexSizes" : {
    "_id_" : 654080,
    "num_1" : 515088
  },
  "ok" : 1
}

```

2.3.2 命令如何执行

MongoDB特定的操作集合——与本章目前为止介绍的insert、update、remove、query命令不同——称为数据库命令。数据库命令通常是管理性的，正如刚才介绍的stats()方法，但是也可以控制MongoDB的核心功能，比如更新数据。

无论提供什么功能，所有的数据库命令实现都有个共同点，就是它们在一个叫做\$cmd的虚拟集合上实现查询。

要理解这句话，我们来快速看个例子，回忆下我们如何调用stats()命令：

```
> db.stats()
```

stats()方法包装了shell命令的方法调用。

可以尝试输入下面等价的操作：

```
> db.runCommand( {dbstats: 1} )
```

执行后的结果与使用stats()方法显示的一样。注意，这个命令也是使用{dbstats: 1}定义的。通常，我们可以通过给runCommand()方法传递参数来调用任意命令。

以下是说明如何运行集合stats命令：

```
> db.runCommand( {collstats: "numbers"} )
```

输出的结果看起来很熟悉。

要掌握数据库命令的内部原理，我们还要看看runCommand()方法是怎么工作的。这个也不难，因为MongoDB shell将会在无括号调用的时候打印任意方法的实现。

如不使用这种运行命令的方式：

```
> db.runCommand()
```

我们可以执行无括号的版本，来看看内部机制：

```
> db.runCommand
function ( obj, extra ) {
  if ( typeof( obj ) == "string" ) {
    var n = {};
    n[obj] = 1;
    obj = n;
    if ( extra && typeof( extra ) == "object" ) {
      for ( var x in extra ) {
        n[x] = extra[x];
      }
    }
  }
  return this.getCollection( "$cmd" ).findOne( obj );
}
```

最后一行就是在\$cmd集合上的查询。恰当的定义为数据库命令就是特殊集合上的查询，\$cmd，查询选择器定义了命令本身。这就是背后原理。你能设想一个手动运行统计命令的方法吗？其实非常简单。

```
> db.$cmd.findOne( {collstats: "numbers"} );
```

使用runCommand帮助方法可以更简单，但是通常最好要理解底层的机制。

2.4 获取帮助

Getting help

目前为止，使用MongoDB shell实验数据操作和管理数据库的价值是显而易见的。但是因为可能大家会在shell花费很多时间，所以要知道如何获取帮助。

内置的帮助命令是首选方式。用db.help()也可以打印通常数据库使用的操作方法。通过运

行`db.numbers.help()`，我们可以找到一个相似的方法列表。

也有`tab`键只能完成功能。开始时输入任意方法的首字母，然后按`tab`键两次，就会看到所有匹配的方法。下面是所有以`get`开头的集合操作方法：

```
> db.numbers.get
db.numbers.getCollection( db.numbers.getIndexes(
db.numbers.getShardDistribution(
db.numbers.getDB( db.numbers.getIndices(
db.numbers.getShardVersion(
db.numbers.getDiskStorageStats( db.numbers.getMongo(
db.numbers.getSlaveOk(
db.numbers.getFullName( db.numbers.getName(
db.numbers.getSplitKeysForChunks(
db.numbers.getIndexKeys( db.numbers.getPagesInRAM(
db.numbers.getWriteConcern(
db.numbers.getIndexSpecs( db.numbers.getPlanCache(
db.numbers.getIndexStats( db.numbers.getQueryOptions(
```

官方的MongoDB手册是最好的资源，我们可以在<http://docs.mongodb.org>找到。它包括教程和参考资料，而且更新到最新的MongoDB版本。文档还包括每个语言的MongoDB驱动实现，比如Ruby驱动，这些驱动在我们使用应用程序访问MongoDB的时候是必须用到的。

如果你还想学习更多内容，而且喜欢使用JavaScript，shell可以让你查看任意的方法实现。例如，假设你想知道`save()`到底是如何工作的。虽然我们可以通过查看MongoDB源码来达到目的，但是还有更简单的方法，就输入方法名，且不需要括号。下面是大家正常执行`save()`方法的命令：

```
> db.numbers.save({num: 123123123});
```

以下就是如何来查看实现代码：

```
> db.numbers.save
function ( obj , opts ){
  if ( obj == null )
    throw "can't save a null";

  if ( typeof( obj ) == "number" || typeof( obj ) == "string" )
    throw "can't save a number or string"

  if ( typeof( obj._id ) == "undefined" ){
    obj._id = new ObjectId();
    return this.insert( obj , opts );
  }
  else {
    return this.update( { _id : obj._id } , obj , Object.merge({
      upsert:true }, opts));
  }
}
```

仔细阅读代码，你就会看到`save()`方法只是对`insert()`和`update()`方法的包装。在检查`obj`参数的类型后，如果要保存的对象没有`_id`字段，就会添加字段，然后调用`insert()`，否则就执行更新操作。

检查shell方法的实现非常方便。记住这个技巧，后面扩展学习MongoDB shell的时候非常有用。

2.5 总结

Summary

现在我们已经实际看到了文档的数据模型，而且我们也演示了数据模型上的不同MongoDB操作。我们已经学习了如何创建索引，以及通过`explain()`基于索引的性能优化。此外，我们已经了解如何查询系统上集合和数据库的信息，也知道了聪明的`$cmd`集合，需要帮助时也知道找到合适的方式。

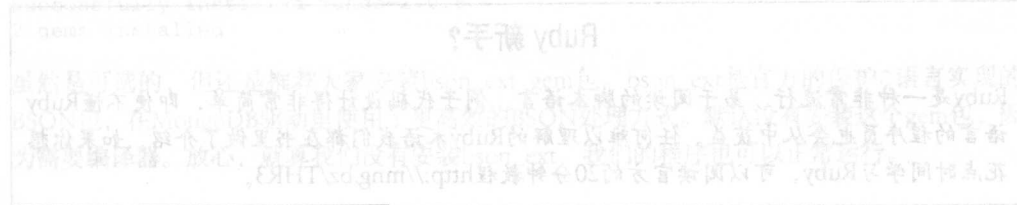
我们学习了许多MongoDB shell的命令，但是这替代不了我们构建真实应用的经验。这也是我们为什么要在下一章里从实验操作转换到真实的数据库开发。我们会学习驱动是如何工作的，然后使用Ruby驱动来构建一个简单的应用，使用真实的数据来操作MongoDB数据库^[1]。

3.1.1 安装与连接

可以使用Ruby on Rails来安装MongoDB驱动，这也可以通过包管理工具来实现。

在本书第1章中，我们介绍了MongoDB的驱动。虽然驱动是存在的，但是驱动的安装和配置却是一个复杂的过程。在本书第2章中，我们将介绍如何安装和配置MongoDB驱动。在本书第3章中，我们将介绍如何使用MongoDB驱动来构建一个简单的应用。在本书第4章中，我们将介绍如何使用MongoDB驱动来构建一个复杂的应用。在本书第5章中，我们将介绍如何使用MongoDB驱动来构建一个分布式应用。在本书第6章中，我们将介绍如何使用MongoDB驱动来构建一个高可用应用。在本书第7章中，我们将介绍如何使用MongoDB驱动来构建一个可扩展应用。在本书第8章中，我们将介绍如何使用MongoDB驱动来构建一个高性能应用。在本书第9章中，我们将介绍如何使用MongoDB驱动来构建一个安全可靠应用。在本书第10章中，我们将介绍如何使用MongoDB驱动来构建一个易于维护应用。在本书第11章中，我们将介绍如何使用MongoDB驱动来构建一个易于部署应用。在本书第12章中，我们将介绍如何使用MongoDB驱动来构建一个易于监控应用。在本书第13章中，我们将介绍如何使用MongoDB驱动来构建一个易于测试应用。在本书第14章中，我们将介绍如何使用MongoDB驱动来构建一个易于集成应用。在本书第15章中，我们将介绍如何使用MongoDB驱动来构建一个易于扩展应用。在本书第16章中，我们将介绍如何使用MongoDB驱动来构建一个易于升级应用。在本书第17章中，我们将介绍如何使用MongoDB驱动来构建一个易于迁移应用。在本书第18章中，我们将介绍如何使用MongoDB驱动来构建一个易于备份应用。在本书第19章中，我们将介绍如何使用MongoDB驱动来构建一个易于恢复应用。在本书第20章中，我们将介绍如何使用MongoDB驱动来构建一个易于灾难恢复应用。在本书第21章中，我们将介绍如何使用MongoDB驱动来构建一个易于合规应用。在本书第22章中，我们将介绍如何使用MongoDB驱动来构建一个易于审计应用。在本书第23章中，我们将介绍如何使用MongoDB驱动来构建一个易于报告应用。在本书第24章中，我们将介绍如何使用MongoDB驱动来构建一个易于分析应用。在本书第25章中，我们将介绍如何使用MongoDB驱动来构建一个易于决策应用。在本书第26章中，我们将介绍如何使用MongoDB驱动来构建一个易于优化应用。在本书第27章中，我们将介绍如何使用MongoDB驱动来构建一个易于改进应用。在本书第28章中，我们将介绍如何使用MongoDB驱动来构建一个易于创新应用。在本书第29章中，我们将介绍如何使用MongoDB驱动来构建一个易于变革应用。在本书第30章中，我们将介绍如何使用MongoDB驱动来构建一个易于未来应用。

在本书第1章中，我们介绍了MongoDB的驱动。虽然驱动是存在的，但是驱动的安装和配置却是一个复杂的过程。在本书第2章中，我们将介绍如何安装和配置MongoDB驱动。在本书第3章中，我们将介绍如何使用MongoDB驱动来构建一个简单的应用。在本书第4章中，我们将介绍如何使用MongoDB驱动来构建一个复杂的应用。在本书第5章中，我们将介绍如何使用MongoDB驱动来构建一个分布式应用。在本书第6章中，我们将介绍如何使用MongoDB驱动来构建一个高可用应用。在本书第7章中，我们将介绍如何使用MongoDB驱动来构建一个可扩展应用。在本书第8章中，我们将介绍如何使用MongoDB驱动来构建一个高性能应用。在本书第9章中，我们将介绍如何使用MongoDB驱动来构建一个安全可靠应用。在本书第10章中，我们将介绍如何使用MongoDB驱动来构建一个易于维护应用。在本书第11章中，我们将介绍如何使用MongoDB驱动来构建一个易于部署应用。在本书第12章中，我们将介绍如何使用MongoDB驱动来构建一个易于监控应用。在本书第13章中，我们将介绍如何使用MongoDB驱动来构建一个易于测试应用。在本书第14章中，我们将介绍如何使用MongoDB驱动来构建一个易于集成应用。在本书第15章中，我们将介绍如何使用MongoDB驱动来构建一个易于扩展应用。在本书第16章中，我们将介绍如何使用MongoDB驱动来构建一个易于升级应用。在本书第17章中，我们将介绍如何使用MongoDB驱动来构建一个易于迁移应用。在本书第18章中，我们将介绍如何使用MongoDB驱动来构建一个易于备份应用。在本书第19章中，我们将介绍如何使用MongoDB驱动来构建一个易于恢复应用。在本书第20章中，我们将介绍如何使用MongoDB驱动来构建一个易于灾难恢复应用。在本书第21章中，我们将介绍如何使用MongoDB驱动来构建一个易于合规应用。在本书第22章中，我们将介绍如何使用MongoDB驱动来构建一个易于审计应用。在本书第23章中，我们将介绍如何使用MongoDB驱动来构建一个易于报告应用。在本书第24章中，我们将介绍如何使用MongoDB驱动来构建一个易于分析应用。在本书第25章中，我们将介绍如何使用MongoDB驱动来构建一个易于决策应用。在本书第26章中，我们将介绍如何使用MongoDB驱动来构建一个易于优化应用。在本书第27章中，我们将介绍如何使用MongoDB驱动来构建一个易于改进应用。在本书第28章中，我们将介绍如何使用MongoDB驱动来构建一个易于创新应用。在本书第29章中，我们将介绍如何使用MongoDB驱动来构建一个易于变革应用。在本书第30章中，我们将介绍如何使用MongoDB驱动来构建一个易于未来应用。



^[1]【译者注】所有语言的官方驱动都可以在官方网站下载，也可以通过包管理工具搜索 MongoDB。无论是 Node.js 还是 Java C# 都可以找到。对于别的语言的例子，大家也可以自己动手实践，参考官方文档即可：<https://docs.mongodb.com/ecosystem/drivers/>。中国 MongoDB 学习交流群 511943641，其中有我写的 Java 和 C# 例子程序。

编写代码操作MongoDB

Writing programs using MongoDB

本章内容

- 通过Ruby介绍MongoDB API
- 理解驱动的工作原理
- 使用BSON和MongoDB网络协议
- 构建完整的例子应用

现在是实战的时候了！虽然实验MongoDB shell还有很多要学习的，但是只有在实际开发过程中使用这个数据库我们才能体会出它的真实价值。这意味着我们要进行编程并且了解MongoDB驱动。正如之前提到的，MongoDB公司提供了针对几乎所有语言的官方支持，Apache许可所有的MongoDB驱动。本书中的例子代码使用了Ruby语言，但是我们演示时使用的方法同样适用于其他语言。本书里我们会使用JavaScript shell演示绝大部分命令，但是通过应用程序访问MongoDB的演示我们将使用Ruby语言。

分三个阶段来学习MongoDB开发编程。第一阶段，学习如何安装MongoDB Ruby驱动，学习基本的CRUD操作。这个过程很快，大家应该也感觉很熟悉，因为API与shell类似。第二阶段会深入学习驱动，了解驱动如何与MongoDB交互。在这个阶段我们将会介绍通常驱动背后的机制。第三阶段，我们通过开发一个简单的Ruby应用来监控Twitter，使用真实的数据集，我们会看到MongoDB如何工作。这个例子也会为第二部分的深入内容奠定基础。

Ruby 新手？

Ruby是一种非常流行、易于阅读的脚本语言。例子代码设计得非常简单，即使不懂Ruby语言的程序员也会从中获益。任何难以理解的Ruby术语我们都在书里做了介绍。如果你想花点时间学习Ruby，可以阅读官方的20分钟教程<http://mng.bz/THR3>。

3.1 通过 Ruby lens 连接 MongoDB

MongoDB through the Ruby lens

通常，当我们思考驱动时，首先想到的就是低级别的二进制数据以及僵硬的接口。幸运的是，MongoDB驱动不是这样设计的。相反，它设计成非常直观、语言敏感的API，许多专业应用都可以使用MongoDB驱动作为连接数据库的唯一接口了。

驱动API跨语言时依然保持一致，这意味着开发者可以方便地选择合适的语言，任意在JavaScript API可以做的事情在Ruby API也都可以做。如果你是应用开发者，就肯定可以找到适合自己的语言的MongoDB驱动，而不需要担心底层的实现细节。

第一节里我们会先安装MongoDB Ruby驱动，连接数据库，并且介绍如何执行基本的CRUD操作。这将为本章结束时开发的应用打下基础。

3.1.1 安装与连接

可以使用RubyGems来安装MongoDB Ruby驱动，这是Ruby的包管理工具^[1]。

许多新操作系统提前预装了Ruby。你也可以通过shell里运行`ruby-version`来检查安装的Ruby。如果没有安装，可以在www.ruby-lang.org/en/downloads下载，其中有详细的安装指南。

我们还需要RubyGems包管理工具。可能也已经安装了。可通过运行`gem-version`来检查。RubyGems的安装指南可以在<http://docs.rubygems.org/read/chapter/3>找到。一旦安装完成，就可以运行命令：

```
gem install mongo
```

这里应该会安装**mongo**和**bson**^[2]的**gems**包，会看到下面的输出信息（新的版本号可能不太一样）：

```
Fetching: bson-3.2.1.gem (100%)
Building native extensions. This could take a while...
Successfully installed bson-3.2.1
Fetching: mongo-2.0.6.gem (100%)
Successfully installed mongo-2.0.6
2 gems installed
```

虽然是可选的，但还是推荐大家安装**bson_ext** gem包。**bson_ext**是官方的保护C语言实现的BSON包，在MongoDB驱动里使用了更高效的BSON处理方式。默认没有安装这个gem包，因为需要编译器。放心，就算我们没有安装**bson_ext**，我们的程序也可以正常运行。

^[1] [译者注] .NET 程序使用 NuGet, Node.js 可以使用 NPM, Java 可以使用 Maven。中国 MongoDB 学习交流群 511943641。

^[2] BSON，在下一节里介绍，是二进制JSON格式，MongoDB用它来表示文档数据。**bson** Ruby gem包会从BSON序列化Ruby对象。

我们会从连接MongoDB数据库开始。首先，通过运行mongo shell来确保已经连接成功运行的mongod进程。接下来，创建一个connect.rb文件，然后输入以下代码：

```
require 'rubygems'
require 'mongo'

$client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'tutorial')
Mongo::Logger.logger.level = ::Logger::ERROR
$users = $client[:users]
puts 'connected!'
```

前两个require语句确保我们可以下载驱动。下面三行初始化客户端连接localhost和tutorial数据库，在\$users变量里存储users集合引用，然后打印connected!字符串！我们在每个变量前面加了\$符号，确保每个变量是全局的，这样可以在connect.rb脚本之外访问、保存文件，然后运行：

```
$ ruby connect.rb
D, [2015-06-05T12:32:38.843933 #33946] DEBUG -- : MONGODB | Adding
  127.0.0.1:27017 to the cluster. | runtime: 0.0031ms
D, [2015-06-05T12:32:38.847534 #33946] DEBUG -- : MONGODB | COMMAND |
  namespace=admin.$cmd selector={:ismaster=>1} flags=[] limit=-1 skip=0
  project=nil | runtime: 3.4170ms
connected!
```

不出意外的话，可以正常通过Ruby连接MongoDB数据库。应该可以在shell里看到connected!字符串。这看起来非常简单，但是确实是连接MongoDB数据库的第一步。接下来，我们会使用连接再插入一些文档数据。

3.1.2 Ruby 里插入文档数据

要运行有趣的MongoDB查询测试，首先要有一些数据，因此我们先来创建一些。设计MongoDB驱动时使用了自己的语言文档表示形式。在Javascript里，JSON对象是个理所当然的选择，因为JSON是文档数据结构；在Ruby里，哈希结构是最有意义的。原生的Ruby哈希与JSON对象只有一些差别，最显著的就是JSON使用冒号来分割键值，而Ruby使用的是哈希符号(=>)^[1]。

如果跟着流程，可以继续往connect.rb文件里添加数据。另外一个方法是直接使用Ruby的shell，Irb。Irb是个REPL(read, evaluate, print loop)控制台，可以输入动态执行的Ruby代码，用于实验操作是非常理想的。Irb编写的任意代码都可以保存到脚本里，所以我们推荐使用它来学习新知识。你也可以启动irb，并且只需要使用connect.rb就可以立即访问链接、数据库和集合对象了，然后就可以运行Ruby代码并接受反馈结果。

以下是个例子：

^[1]Ruby 1.9 里，也可以使用冒号分割键值，像哈希 hash = {foo: 'bar'} 一样，但是为了兼容性我们还是坚持使用哈希符号。

```
$ irb -r ./connect.rb
irb(main):017:0> id = $users.insert_one({"last_name" => "mtsouk"})
=> #<Mongo::Operation::Result:70275279152800 documents=[{"ok"=>1, "n"=>1}]>
irb(main):014:0> $users.find().each do |user|
irb(main):015:1* puts user
irb(main):016:1> end
{"_id"=>BSON::ObjectId('55e3e1c5ae119511d000000'), "last_name"=>"knuth"}
{"_id"=>BSON::ObjectId('55e3f13d5ae119516a000000'), "last_name"=>"mtsouk"}
=> #<Enumerator: #<Mongo::Cursor:0x70275279317980
@view=#<Mongo::Collection::View:0x70275279322740 namespace='tutorial.users
@selector={} @options={}>>:each>
```

irb的命令行shell带有> (不同的机器看起来可能不一样)。它允许我们输入不同的命令, 以及之前代码里我们高亮的命令。当我们在irb里运行命令时, 如果有数据, 它就会返回命令的输出结果。这就是=>符号后显示的东西。

我们来为users集合添加一些文档数据。我们可以创建2个文档表示2个用户, smith 和 jones。每个文档使用Ruby hash哈希表示, 赋值给变量:

```
smith = {"last_name" => "smith", "age" => 30}
jones = {"last_name" => "jones", "age" => 40}
```

要保存文档, 需要把对象传递给集合的insert方法。每次调用insert都会返回我们可以保存到变量里便于后期查询使用的唯一ID:

```
smith_id = $users.insert_one(smith)
jones_id = $users.insert_one(jones)
```

我们可以使用简单的查询来检验保存的文档数据, 所以可以使用user集合的find() 方法进行查询, 如下所示:

```
irb(main):013:0> $users.find("age" => {"$gt" => 20}).each.to_a do |row|
irb(main):014:1* puts row
irb(main):015:1> end
=> [{"_id"=>BSON::ObjectId('55e3f7dd5ae119516a000002'),
"last_name"=>"smith",
"age"=>30}, {"_id"=>BSON::ObjectId('55e3f7e25ae119516a000003'),
"last_name"=>"jones", "age"=>40}]
```

如果运行irb, 命令窗口里会显示查询结果。如果从Ruby文件里运行代码, 则前置Ruby的p方法用来打印输出信息:

```
p $users.find( :age => {"$gt" => 20}).to_a
```

我们已经成功地从Ruby插入了2条文档数据。现在来详细看一下查询功能。

3.1.3 查询与光标

既然我们已经创建了文档, 现在应该来学习MongoDB提供的查询操作了 (CRUD的R)。Ruby驱动定义了丰富的接口来访问数据, 而且处理了许多底层的细节。本节里展示的查询是非常

简单的查询，其实MongoDB允许我们执行更加复杂的查询，比如文本搜索和聚合，这在后面的章节里会介绍。

我们会从标准的find方法来看这个如何实现。这里是数据集上两个可能的查找操作：

```
$users.find({"last_name" => "smith"}).to_a
$users.find({"age" => {"$gt" => 30}}).to_a
```

第一个查询last_name是smith的用户文档，第二个查询所有年龄age大于30的用户文档。尝试在irb输入第三个查询：

```
2.1.4 :020 > $users.find({"age" => {"$gt" => 30}})
=> #<Mongo::Collection::View:0x70210212601420 namespace='tutorial.users'
@selector={"age"=>{"$gt"=>30}} @options={} >
```

结果在Mongo::Collection::View对象里返回，它扩展自Iterable接口，方便迭代结果集合。我们会在3.2.3一节里详细讨论这个问题。同时，我们也可以获取\$gt查询的结果：

```
cursor = $users.find({"age" => {"$gt" => 30}})
cursor.each do |doc|
  puts doc["last_name"]
end
```

这是Ruby的 each迭代器，传递每个结果给代码块。last_name被打印在控制台上。查询中使用的\$gt是个MongoDB操作符，\$字符与Ruby里定义全局变量的Ruby没有关系，比如\$users。如果集合里文档没有last_name字段，你可能发现Ruby会打印nil（Ruby的null值），这表示缺少值，而且很常见。

从上一章shell的例子，你可能要考虑的是光标。shell使用光标的方式与其他驱动基本一样，其不同点在于shell会自动迭代find()结果里的20个光标。要获取其他的结果，可以继续使用it命令手动迭代。

3.1.4 更新和删除

回忆一下第2章里的updates，至少需要2个参数：查询选择权和更新文档。下面是使用Ruby驱动的例子：

```
$users.find({"last_name" => "smith"}).update_one({"$set" => {"city" =>
"Chicago"}})
```

这个更新会首先找到last_name是smith的文档，如果找到就会把city设置为Chicago。这个更新使用了\$set操作符。你可以运行查询来查看修改：

```
$users.find({"last_name" => "smith"}).to_a
```

该视图允许我们决定只更新一个文档还是更新匹配查询的所有文档。在之前的例子里，就算你有几个名字为smith的文档，也只有一个会被更新。要为特别的smith更新数据，就需要添加

更多的查询选择器。如果你想更新所有的smith文档，就必须使用update_many替换update_one方法：

```
$users.find({"last_name" => "smith"}).update_many({"$set" => {"city" => "Chicago"}})
```

删除数据更加简单。我们已经讨论了MongoDB shell是如何工作的，与Ruby驱动也没有什么不同。复习下：只需要使用remove方法。这个方法接受一个选择器参数，只删除匹配的文档数据。如果不提供参数，就会删除集合里的所有数据。以下例子假设要删除所有年龄大于40岁的用户文档：

```
$users.find({"age" => {"$gte" => 40}}).delete_one
```

这里只会删除匹配查询的第一个数据。如果要删除所有匹配的文档，就需要运行：

```
$users.find({"age" => {"$gte" => 40}}).delete_many
```

无参数时，drop方法会删除所有剩余的文档：

```
$users.drop
```

3.1.5 数据库命令

在前面一章，我们已经学习了数据库命令。我们看到了2个stats命令。这里，我们将会学习如何在驱动里使用listDatabases运行命令。这是管理数据库必须运行的命令，它在启动验证的时候会被特殊对待。关于验证和管理数据库的详细内容，请参考第10章。

首先，我们创建一个Ruby数据库对象来引用admin数据库。然后传递查询命令给command方法：

```
$admin_db = $client.use('admin')
$admin_db.command({"listDatabases" => 1})
```

注意：这些代码仍然会用到我们在connect.rb脚本里保存的代码，因为它需要使用\$client里保存的连接。应答结果使用Ruby哈希，列出了所有的数据库以及磁盘大小：

```
#<Mongo::Operation::Result:70112905054200 documents=[{"databases"=>[
  {
    "name"=>"local",
    "sizeOnDisk"=>83886080.0,
    "empty"=>false
  },
  {
    "name"=>"tutorial",
    "sizeOnDisk"=>83886080.0,
    "empty"=>false
  },
  {
    "name"=>"admin",
```

```
"sizeOnDisk"=>1.0, "empty"=>true
}}, "totalSize"=>167772160.0, "ok"=>1.0]]>
=> nil
```

这可能与`irb`和MongoDB驱动不同，但是应该很容易访问。一旦适应了Ruby哈希表示的文档，shell API的转换应该是无缝的。

绝大部分驱动都提供了和数据库命令一致的功能接口。你可能还记得前一章里介绍的，用`remove`命令不会真正删除集合。要删除集合和索引，就必须使用`drop_collection`方法：

```
db = $client.use('tutorial')
db['users'].drop
```

如果对于使用MongoDB的Ruby驱动还不够熟练，那也没有关系，我们会在3.3节里获得更多练习机会。但是现在我们将会花点时间来学习MongoDB驱动的工作原理。这将会揭示更多的MongoDB设计原理，方便大家更高效地使用驱动。

3.2 驱动工作原理

How the drivers work

现在是时候明白在驱动或者MongoDB shell里输入命令后发生的故事了。本节里，我们将会看到驱动序列化数据并与数据库通信。

所有的MongoDB驱动都执行三个主要的功能：首先，生成MongoDB对象ID。默认都存储在所有的文档的`_id`字段里。其次，驱动会把任意语言表示的文档对象转换为BSON或者从BSON转换回来，BSON是MongoDB使用的二进制JSON格式。之前的例子里，驱动序列化所有的Ruby哈希为BSON，或者把BSON转换为Ruby哈希。

最后，使用TCP socket与数据库连接通信，此时使用的是MongoDB自定义协议。协议的详细内容不在本节的讨论范围里。socket通信的风格，特别是写入数据等待应答是非常重要的，我们将会在本节里详细介绍。

每个MongoDB文档都需要一个主键。这个键对于每个集合里的文档来说必须是唯一的，存储在文档的`_id`字段里。开发者也可以使用自己的值作为`_id`的值，但是没有提供值时，MongoDB会使用自己的默认值。在发送文档数据给服务器之前，驱动会检查是否有`_id`字段。如果没有，就会生产一个对象`_id`。

MongoDB对象ID被设计成全局唯一，这意味着它可以在特定的上下文里确保唯一。如何确保呢？我们来详细看一下算法。

如果你仔细研究过MongoDB，就应该见过对象ID。首先，它看起来很像一串随机文本，比如4c291856238d3b19b2000001。你可能没有注意到，这串字符是12个字节的十六进制形式。它

们确实保存着一些有用的信息。这些字节有特殊的结构，如图3.1所示。

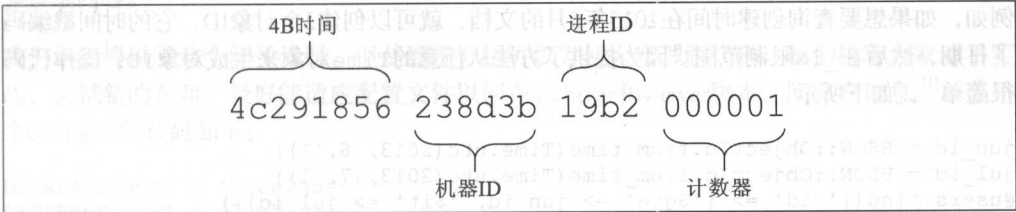


图 3.1 MongoDB 对象 ID 格式

最重要的4个字节包含着标准的Unix时间戳^[1]。后面3个字节是机器ID，紧接着是2个字节的进程ID。最后3个字节存储的是进程本地计数器，每次生成新ID就会自动增长。计数器可以保证同一个进程和同一时刻内不会重复。

为什么对象ID有这种格式？很重要的原因是ID是在驱动里生成的，而不是在服务器上生成的。这与许多RDBMS系统不同。关系型数据在服务器端自增主键，因此导致服务器成为生成键的瓶颈。如果多个驱动生成ID并插入文档，它们就需要创建唯一标识符而不会互相影响的方法。因此，时间戳、机器ID和进程ID包含在标识符中，让ID无法重复。

你可能已经考虑了这种问题发生的概率。实际上，在遇到计数器（每秒2²⁴百万次）瓶颈之前已经遇到了插入文档的瓶颈了。可以稍微想象一下（不太可能），如果你跨不同的机器分布式部署驱动，就不可能有相同的机器ID。例如，Ruby驱动使用下面的机器ID计算公式：

```
@@machine_id = Digest::MD5.digest(Socket.gethostname)[0, 3]
```

对于可能的问题，它们可能使用相同的进程ID启动MongoDB驱动，而且某个时刻有相同的计数值。

实际上，不要担心重复，因为几乎是不可能的。

使用MongoDB对象ID一个附加的好处就是它带有时间戳。绝大部分驱动都允许我们获取时间戳，因此提供了文档的创建时间，可以解析出最近的秒数。

使用Ruby驱动，也可以通过调用对象ID的generation_time方法来获取ID的创建时间，作为Ruby Time对象。

```
irb> require 'mongo'
irb> id = BSON::ObjectId.from_string('4c291856238d3b19b2000001')
=> BSON::ObjectId('4c291856238d3b19b2000001')
irb> id.generation_time
=> 2010-06-28 21:47:02 UTC
```

^[1]许多 UNIX 机器（也包括 Linux）存储时间的格式为 UNIX Time 或者 POSIX time 格式，它们只从 1970 年 1 月 1 号 00:00 开始计数。这意味着时间戳可以是整数。例如，2010-06-28 21:47:02 可以表示为 1277761622（或者十六进制的 0x4c291856），从开始 epoch 到现在的总秒数。

自然而然地，你也可以使用对象ID来确定对象创建时间的查询范围。

例如，如果想要查询创建时间在2013年6月的文档，就可以创建2个对象ID，它的时间戳编码了日期，然后在`_id`限制范围。因为提供了方法从任意的Time对象来生成对象ID，操作代码很简单^[1]，如下所示：

```
jun_id = BSON::ObjectId.from_time(Time.utc(2013, 6, 1))
jul_id = BSON::ObjectId.from_time(Time.utc(2013, 7, 1))
@users.find({'_id' => {'$gte' => jun_id, '$lt' => jul_id}})
```

正如前面提到的，你也可以设置`_id`的值。当文档的某个字段十分重要，而且一直唯一时，这个可以起作用。例如，`users`集合要存储`username`在`_id`字段里，而不是在对象ID中。这两种方法各有好处，可以依开发者的个人喜好来选择。

3.3 构建简单的应用

Building a simple application

接下来我们会构建一个简单的应用来存档和显示Tweets内容。可以把这个当做一个大型网站的组件，允许用户监控和保留与他们业务相关的搜索词语。这个例子会告诉大家，调用Twitter API后返回的是JSON数据并转换为MongoDB文档是多么简单、方便。如果使用关系型数据库，必须提前设计数据schema，可能由许多表组成，而且要提前定义这些表。这里我们什么都不需要做，只需要保存有丰富数据结构的Tweet文档，而且要高效地查询它们。

我们把这个项目叫做TweetArchiver。TweetArchiver有2个组件：Archiver和viewer。Archiver会调用Twitter search API并保存相关的推文，viewer会在Web浏览器里显示结果。

3.3.1 设置

这个应用需要4个Ruby库。本章的代码存储库包含名为Gemfile的文件，它列出了这些gems包。

把工作目录修改为chapter3，然后确保`ls`命令显示Gemfile。然后使用如下命令进行安装：

```
gem install bundler
bundle install
```

这会确保安装bundler gem。接下来，使用Bundler包管理工具安装其他的gem。在Ruby里经常这么做，可以自动匹配需要的版本：我们例子代码需要的版本库。

我们的Gemfile 列举了 mongo、twitter、bson、sinatra等4个gem，所以都会安装。我们已经使用了mongo gem，这里包含它是为了确保正确的版本。twitter gem用来和Twitter

^[1]例子实际上无法工作，只是个详细的练习。现在你应该可以有足够的知识来创建有意义的数据进行查询练习了。为什么不开始动手呢？

API进行通信。sinatra gem是运行Ruby网站的简单Web服务器，我们将会在3.3.3里讨论更多详细内容。

我们单独提供了这个例子代码，但是逐步介绍可以方便大家理解代码。我们推荐大家动手实战，尝试新的东西。最好能适应配置文件以便在archiver和viewer脚本之间共享信息。创建一个config.rb的代码如下：

```
DATABASE_HOST = 'localhost'
DATABASE_PORT = 27017
DATABASE_NAME = "twitter-archive"
COLLECTION_NAME = "tweets"
TAGS = ["#MongoDB", "#Mongo"]

CONSUMER_KEY = "replace me"
CONSUMER_SECRET = "replace me"
TOKEN = "replace me"
TOKEN_SECRET = "replace me"
```

首先我们指定了网站要使用的数据库和集合的名字。然后我们定义了搜索词汇的可能要发送给Twitter API的数组。

Twitter需要用户注册一个免费账号才能访问API。大家可以在<http://apps.twitter.com>注册。一旦注册了应用，就可以看到它的验证信息页面，也可能在API keys选项卡里。我们会点击按钮，创建访问令牌(token)。我们会使用这些参数来调用后台的API。

3.3.2 搜集数据

接下来是要编写archiver脚本。我们可以从TweetArchiver开始。我们使用搜索关键字来初始化这个类。然后调用TweetArchiver实例的update方法，它会发起API调用。最后把结果保存到MongoDB集合里。

我们来看看此类构造函数：

```
def initialize(tag)
  connection = Mongo::Connection.new(DATABASE_HOST, DATABASE_PORT)
  db = connection[DATABASE_NAME]
  @tweets = db[COLLECTION_NAME]
  @tweets.ensure_index([['tags', 1], ['id', -1]])
  @tag = tag
  @tweets_found = 0

  @client = Twitter::REST::Client.new do |config|
    config.consumer_key = API_KEY
    config.consumer_secret = API_SECRET
    config.access_token = ACCESS_TOKEN
    config.access_token_secret = ACCESS_TOKEN_SECRET
  end
end
```


Initialize方法创建了一个连接、数据库和集合对象，我们用来存储Tweets数据。

可以在tags升序和id降序上创建一个复合索引。因为我们想根据tag进行搜索，然后从新到旧来排序显示，tags升序和id降序的索引会让查询结果基于索引进行过滤。正如你看到的，1表示升序，-1表示降序。不要担心，我们会在第8章里深入讨论这些知识。

我们还需使用config.rb的验证信息来配置Twitter客户端。这一步会传递这些值给Twitter gem，它会使用这些参数来调用Twitter API。Ruby对于这种配置有特殊的语法；config变量传递给Ruby模块，我们可以在这里设置配置信息。

MongoDB允许我们在不知道数据结构的情况下插入数据。而关系型数据库需要一个提前定义好的schema，要提前知道要存储的数据。将来Twitter可能通过修改API来返回不同的数据，可能需要修改schema才能存储不同的数据。使用MongoDB，就无须担心，MongoDB使用无schema模式来存储Twitter API返回的数据。

Ruby Twitter库返回一个Ruby哈希对象，所以我们可以直接给MongoDB集合对象传递这个对象。在TweetArchiver代码里，我们可以直接添加如下实例方法：

```
def save_tweets_for(term)
  @client.search(term).each do |tweet|
    @tweets_found += 1
    tweet_doc = tweet.to_h
    tweet_doc[:tags] = term
    tweet_doc[:id] = tweet_doc[:id]
    @tweets.insert_one(tweet_doc)
  end
end
```

在保存每个Tweet文档之前，要做两个小修改。为了简化查询，给每个文档对象添加一个tags属性。也可以为每个对象设置 id字段值作为文档的ID，替换集合的主键，确保每个Tweet推文对象都是唯一的。然后把修改过的文档对象传递给save方法。

要在类里使用这些代码，还需要做一些工作。首先，必须配置MongoDB驱动，这样才可以连接正确的mongod服务，使用正确的数据库和集合。当我们使用MongoDB的时候，这个代码非常简单，我们可以直接复制。其次，必须为Twitter gem配置开发者账号。

这一步是必须的，因为Twitter限制只有注册的开发人员才可以调用API。列表3.1里也提供了update方法，它提供用户反馈和调用。

列表3.1 获取Tweet推文并保存到MongoDB数据库的类中

```
$LOAD_PATH << File.dirname(__FILE__)
require 'rubygems'
require 'mongo'
require 'twitter'
require 'config'

class TweetArchiver
```

```

def initialize(tag)
  client =
    Mongo::Client.new(["#{DATABASE_HOST}:#{DATABASE_PORT}"], :database =>
      "#{DATABASE_NAME}")
  @tweets = client["#{COLLECTION_NAME}"]
  @tweets.indexes.drop_all
  @tweets.indexes.create_many([
    { :key => { tags: 1 } },
    { :key => { id: -1 } }
  ])
  @tag = tag
  @tweets_found = 0

  @client = Twitter::REST::Client.new do |config|
    config.consumer_key = "#{API_KEY}"
    config.consumer_secret = "#{API_SECRET}"
    config.access_token = "#{ACCESS_TOKEN}"
    config.access_token_secret = "#{ACCESS_TOKEN_SECRET}"
  end
end

def update
  puts "Starting Twitter search for '#{@tag}'..."
  save_tweets_for(@tag)
  print "#{@tweets_found} Tweets saved.\n\n"
end

private
def save_tweets_for(term)
  @client.search(term).each do |tweet|
    @tweets_found += 1
    tweet_doc = tweet.to_h
    tweet_doc[:tags] = term
    tweet_doc[:_id] = tweet_doc[:id]
    @tweets.insert_one(tweet_doc)
  end
end
end

```

创建
Tweet-
Archive
的实例

使用 config.rb
中的值配置

保存 save_tweets_for
的方法

使用 Twitter 搜索
并保存结果到
Mongo

剩下的代码就是编写脚本来运行TweetArchiver代码，根据每次输入的搜索关键字来搜索结果。我们可以创建一个名为update.rb的文件（或者从提供的代码里复制），它包含的代码如下：

```

$LOAD_PATH << File.dirname(__FILE__)
require 'config'
require 'archiver'
TAGS.each do |tag|
  archive = TweetArchiver.new(tag)
  archive.update
end

```

接下来，运行更新的脚本：

```
ruby update.rb
```

我们会看到一些提示Tweet推文是否找到和保存的信息。我们也可以通过MongoDB shell直接查询集合来检查脚本是否可以工作：

```
> use twitter-archive
switched to db twitter-archive
> db.tweets.count()
30
```

这里最重要的是我们使用几行代码来管理Twitter搜索结果的推文保存过程^[1]。接下来就是现实结果的代码了。

3.3.3 查看存档

我们可以使用Ruby的Sinatra Web框架来构建一个简单的App来显示结果。

Sinatra允许直接定义Web应用的终结点(endpoint)，以及指定应答消息。它的强大之处依赖于其简易性。例如，index页面的内容可以设置如下：

```
get '/' do
  "response"
End
```

这个代码指定了/终结点的get请求返回response的值给客户端。使用这个格式，你可以编写许多终结点给Web程序，每个都可以在返回结果之前执行特定的Ruby代码。我们也可以发现更多的信息，包括Sinatra完整文档，参见<http://sinatrarb.com>。

我们现在引入一个新的文件viewer.rb，与其他脚本放在同一个目录下。接下来，创建个新的文件夹叫views，然后创建个文件叫tweets.erb。完成这些工作以后，项目的目录结构应该如下：

```
- config.rb
- archiver.rb
- update.rb
- viewer.rb
- /views
  - tweets.erb
```

再说一次，大家可以自己创建文件或者直接从例子代码文件里复制文件。

■ 现在编写viewer.rb文件，使用如表3.2的代码。

列表3.2 显示存档推文的Sinatra应用

```
$LOAD_PATH << File.dirname(__FILE__)
require 'rubygems'
require 'mongo'
require 'sinatra'
require 'config'
require 'open-uri'
```

① 必备库

^[1]也可以使用更少的代码实现。这个就作为练习留给读者了。

```

configure do
  client = Mongo::Client.new(["#{DATABASE_HOST}:", :database
    => "#{DATABASE_NAME}"])
  TWEETS = client["#{COLLECTION_NAME}"]
end

get '/' do
  if params['tag']
    selector = {:tags => params['tag']}
  else
    selector = {}
  end

  @tweets = TWEETS.find(selector).sort(["id", -1])
  erb :tweets
end

```

实例化集合

动态构建 jQuery 选择器

或者使用空选择器

渲染视图

查询

第一行需要必备的库，与配置文件一致①。接下来，有个配置块创建MongoDB连接并存储tweets集合的引用到TWEETS里②。

真正核心的代码就是get '/' do开始的部分。这个部分的代码处理应用根目录URL的请求。首先，我们创建查询选择器。如果提供了URL Tag标签参数，就要创建一个查询选择器来限制结果集③。否则就创建空选择器，它会返回集合里的所有文档④。然后执行查询⑤。目前为止，我们应该知道@tweets变量里赋值的是个光标而不是结果集。我们还需要在视图里迭代循环处理光标。

倒数第二行⑥渲染视图文件tweets.erb（参见列表3.3）。

列表3.3 嵌入Ruby脚本的HTML代码来显示推文信息

```

<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <style>
    body {
      width: 1000px;
      margin: 50px auto;
      font-family: Palatino, serif;
      background-color: #dbd4c2;
      color: #555050;
    }
    h2 {
      margin-top: 2em;
      font-family: Arial, sans-serif;
      font-weight: 100;
    }
  </style>
</head>
<body>
<h1>Tweet Archive</h1>
<% TAGS.each do |tag| %>
  <a href="/?tag=<%= URI::encode(tag) %>"><%= tag %></a>
<% end %>

```

```
<% @tweets.each do |tweet| %>
  <h2><%= tweet['text'] %></h2>
  <p>
    <a href="http://twitter.com/<%= tweet['user']['screen_name'] %>">
      <%= tweet['user']['screen_name'] %>
    </a>
    on <%= tweet['created_at'] %>
  </p>
  
<% end %>
</body>
</html>
```

绝大部分是HTML代码，混合了部分ERB（嵌入的Ruby代码）。Sinatra app 通过ERB处理器，运行tweets.erb文件，并评估<% 和 %>符号之间的Ruby代码。

最重要的部分就出现了，使用2个迭代器。第一个循环通过tags列表来展示某个tag的超链接。

第二个循环，使用@tweets.each代码，循环显示每个Tweet的文本，创建日期和用户头衔。我们可以通过运行程序来查看结果：

```
$ ruby viewer.rb
```

如果启动程序无错，我们会看到标准的Sinatra启动消息，结果如下：

```
$ ruby viewer.rb
[2013-07-05 18:30:19] INFO WEBrick 1.3.1
[2013-07-05 18:30:19] INFO ruby 1.9.3 (2012-04-20) [x86_64-darwin10.8.0]
== Sinatra/1.4.3 has taken the stage on 4567 for development with backup from WEBrick
[2013-07-05 18:30:19] INFO WEBrick::HTTPServer#start: pid=18465 port=4567
```

我们可以把浏览器地址指向http://localhost:4567。页面截图应该如图3.2所示。尝试点击屏幕顶部的链接来限制搜索特定的tag标签。

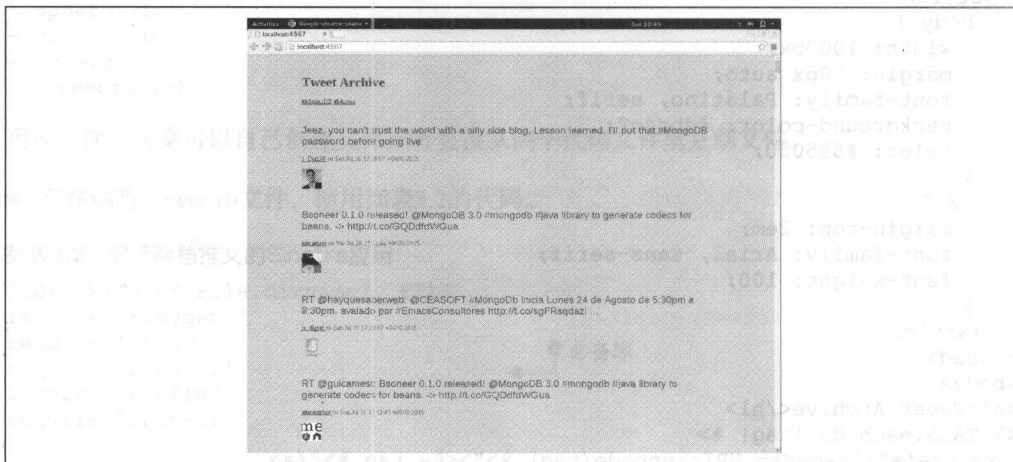


图 3.2 浏览器里渲染的 Tweet Archiver 存档数据

这是应用的扩展部分。确实很简单，但是它只演示了使用MongoDB多么简单。我们无须提前定义数据结构schema，而且可以使用辅助索引加速查询，阻止重复的插入，而且使用编程语言调用MongoDB也非常简单。

3.4 总结

Summary

我们刚刚已经学习了如何使用Ruby语言来操作MongoDB数据库。我们也看了Ruby表示文档数
据多么简单，而且CRUD操作的API与shell的几乎一样。我们也深入学习了驱动内部关于对象
ID、BSON以及MongoDB网络协议的知识。最后，我们也构建了一个简单的MongoDB应用程
序来编写真实的代码。虽然真实的项目使用MongoDB时要复杂得多，但是开发应用程序操作
数据库的方式应该是相似的。

在第4章里，我们将会用到目前为止学习的一切知识。特别是，我们将会实战开发使用
MongoDB构建一个电子商务网站。这是个非常庞大的项目，所以我们将关注以后开发的某
些功能模块。我们将会介绍该领域的某些数据模型，而且会介绍如何插入和查询该类型的数
据。

第1章，我们会学习一些MongoDB的设计原则，然后会简单展示如何创建用户、订单和商品评分数
据。本章包含BSON的核心数据类型。

第2章涵盖了MongoDB的查询语言。我们会学习如何基于一些介绍的数据模型使用查询语言
查询一些数据。此外，我们会学习到如何插入和删除数据。

第3章是关于索引设计的内容。我们会介绍如何索引一些简单的字段，并且会展示索引
MongoDB的索引类型。

在介绍MongoDB的更新和删除操作时，第4章通过介绍电子商务数据模型的设计来展示这
一部分的知识。我们不会深入学习如何维护索引层级和如何管理索引。此外，还将介
绍强大的更新操作符Update和FindAndModify命令。

第二部分 MongoDB 应用系统开发

Application development in MongoDB

本书的第二部分会详细介绍MongoDB的文档数据模型、查询语言和CRUD（新增、读取、更新和删除）操作。

我们会通过逐步实现一个电商数据模型和实现必要的CRUD操作来说明这些问题。每章都会使用自上而下的方式来阐述主题，首先是通过例子介绍电商应用的领域，然后逐步丰富细节。在第一次阅读本书的时候，可能你想只看例子部分的内容，细节留着以后阅读。

第4章，我们会学习一些schema架构设计原则，然后为商品、类别、用户、订单和商品评价构建一个基本的电商数据模型。最后我们会学习MongoDB如何在数据库、集合和文档级别组织数据。本章包含BSON的核心数据类型。

第5章涵盖了MongoDB的查询语言。我们会学习如何基于上一章介绍的数据模型使用常见的查询。在要点小节里，我们会看到查询操作符语义的详细介绍。

第6章是关于聚合概念的内容。我们会介绍如何实现一些简单的分组，并且会深入介绍MongoDB的聚合框架。

在介绍MongoDB的更新和删除操作时，第7章通过介绍电子商务数据模型的设计原理来完善本部分的知识点。我们还会深入学习如何维护类别层级和如何管理仓库事务。此外，还详细介绍强大的更新操作符findAndModify命令。

4.1 schema 设计原则

Principles of schema design

数据库schema设计是基于数据库特性、数据操作和应用系统逻辑等可预见的形式进行。关系型数据库schema设计的原则性更强。对于NoSQL数据库，设计时需要遵守一些原则。

面向文档的数据

Document-oriented data

本章内容

- schema设计
- 电子商务数据模型
- 数据库、集合和文档

本章将会详细看一下面向文档的数据模型以及如何在MongoDB数据库、集合和文档级别里组织这些数据。我们将从一个简单的、通用的案例讨论开始，介绍如何使用MongoDB来设计schema。

记住，MongoDB本身并不强制使用schema，但是每个应用都需要一些如何存储数据的基本内部标准。本章的第二部分会探讨设计原则，检查MongoDB的电商网站schema设计。这样，我们将会看到RDBMS schema和MongoDB的不同，也会学习一些典型的实体关系，比如如何在MongoDB里体现一对多和多对多关系。电商网站的schema展示的知识也是本章后续内容查询、聚合以及更新知识的基础。

因为文档是MongoDB的原材料，所以我们会在本章最后部分详细讲解大家在思考schema的时候可能遇到的问题。这里会更深入地讨论数据库集合和文档知识。如果阅读到结尾，你可能会明白很多特性以及MongoDB文档的限制。大家在阅读本章最后一节的时候可能会发现许多宝藏，因为这里包含了大家使用MongoDB开发系统过程中遇到的许多典型的问题及其答案。

4.1 schema 设计原则

Principles of schema design

数据库schema设计是基于数据库特性、数据属性和应用系统选择最好的数据表示形式的过程。关系型数据库schema设计的原则已经建立了。对于RDBMS数据库，我们只需要遵守数据库设

计范式^[1]即可，它可以帮助我们确保通用查询以及数据一致性。此外，这些原则模式避免开发人员思考如何建模，比如一对多以及多对多的关系。但是schema设计即使对于数据库也并非是一门精确的科学。应用功能和性能是schema设计最重要的考虑因素，所以每个规则都有例外。

如果你来自RDBMS世界，可能会对于MongoDB缺少强制的schema设计原则感到困惑。优秀的实践原则已经出现，但是还有很多好方法可以处理数据集建模问题。本节的原则可以驱动schema的设计，但是现实中这些原则也是弹性的，大家可以灵活选择。

当使用数据库系统建模时，可以先思考以下问题：

- 应用访问的模式是什么？你需要分解需求，不仅仅是落实schema设计，还有选择采用什么数据库。记住，MongoDB并非适用于所有的应用。理解应用的访问模式是目前为止schema设计最重要的方面。

应用程序的特征很容易要求schema严格遵守数据建模的原则，导致你在决定理想的数据模型之前必须问许多问题：读/写的比率是多少？查询是不很简单？查询一个key还是更复杂的key？是否需要聚合查询？数据量是多少？

- 数据的基本单位是什么？在RDBMS里，我们有列和行的表。在键值数据库里，我们有键指向不同的值。在MongoDB里，数据的基本单位是BSON文档。

- 数据库的功能是什么？一旦明白基本的数据类型，就知道如何来操作它了。RDBMS功能 ad hoc查询以及连接通常写入SQL，而简单的键值存储允许通过key获取数据。MongoDB也允许ad hoc查询，但是不支持join连接查询。

数据库更新数据的方式也不同。RDBMS允许使用SQL进行复杂的更新，可以在事务里包含多个更新并支持原子性和回滚。MongoDB不支持事务，但是它支持另外一个原子更新操作，可以更新复杂结构的文档数据。使用键值库，你可以更新一个值，但是每次更新都意味着完全替换一个值。

- 如何记录生成好的唯一ID或者主键？虽然也有例外，但是无论什么数据库系统，许多schema都有记录的唯一key。选择key的策略会影响如何访问和存储数据。如果你正在设计user集合，例如，使用任意值、名字、username或者社会安全号作为主键，那么结果证明在数据集里姓名和社会安全账号都不是唯一的。

MongoDB选择_id字段里存储的值作为主键。这个自动生成的默认值不错，当然并非适用于所有的情况。你要在多个机器上分片存储数据时，这非常重要，因为它决定了数据文档存储在哪个地方。我们会在12章里详细讨论这个问题。

最好的schema设计通常是深入了解使用的数据库、了解应用系统的需求，以及具有丰富的经验之后的产物。好的schema通常需要试验和迭代，比如当应用伸缩时，或者性能考虑变化时。当学习新知识的时候不要担心会修改schema，因为几乎不可能在实现代码之前把所有问题都

^[1]简单理解范式的方式是，信息不会存储2次以上。因此，实体的一对多关系通常会分割为2个表。

了解清楚了。本章的例子用来帮助大家开发好的MongoDB schema设计。学习了这些例子后，我们就可以开始为我们的应用系统设计最好的schema了。

4.2 设计电商网站数据模型

Designing an e-commerce data model

第3章提供的Twitter应用程序例子主要是演示了基本的MongoDB特性，但是不需要关注schema设计。我们在后续的章节里，会学习更多的电商平台领域知识。电子商务的优势是包含大量的、熟悉的数据建模模式，而且，很容易在RDBMS关系型数据库里进行products、categories、product reviews、orders等建模。这个可以让例子更容易理解，因为我们可以与之前先入为主的schema设计规则进行对比。

电商网站使用RDBMS数据库有几个原因。首先，电商网站通常需要事务，这是关系型数据库的强项。其次，需要复杂数据模型和复杂查询的领域最适合采用关系型数据库。

下面的例子可以说明第二个观点。

开发完整的电商后台并非本书的主要内容。相反，我们将会处理一些常见的、有用的电商实体，比如产品和客户评价，以及如何在MongoDB里建模。尤其是，我们将会看到一些产品、类别、用户、订单、商品评价。对于每个实体，我们将会展示例子文档。然后，将会展示一些数据库特性、完善文档的结构。

对于许多开发者来说，数据模型和对象映射一样重要，因此他们可能会使用一些ORM框架，必须是Java的Hibernate或者Ruby的ActiveRecord。这些框架可以和RDBMS关系型数据库高效地结合，但是无法与MongoDB一起使用。这是因为文档本身已经是类对象表示形式了。其次是MongoDB驱动的原因，它已经提供了对MongoDB的高级别抽象了。毫无疑问，我们只使用驱动接口来构建MongoDB应用。

对象映射器可以通过验证、类型检查和模型关联来提供值，而且都有标准的框架，比如Ruby on Rails。对象映射器也在程序员和数据库之间引入了新的复杂性，隐藏了重要的查询特性。当使用对象映射器时，就应该评估一下它的利与弊。许多优秀的应用，有的使用了对象映射器，有的没有使用对象映射器^[1]。我们在本书的例子中没有使用对象映射器，而且我们推荐第一次使用MongoDB的时候不要使用它。

4.2.1 schema 基础知识

商品和类别是所有电商网站的主要部分。对商品，可以在RDBMS里进行范式模型建模，需要许多表，有基本的商品信息表，比如名字和SKU，而且还有其它表，比如快递信息和价格历

^[1]要找到合适的对象映射器，可以参考 mongodb.org 网站。

史。多表schema可以使用RDBMS的多表关联。

MongoDB建模商品表时简单得多。因为集合不强制使用schema，任意文档都可以满足商品动态字段的需求。通过在文档里定义数组就可以在MongoDB集合里实现RDBMS的多表关联关系。

列表4.1展示了一个园艺店商品的具体文档例子。建议在使用db.products.insert(yourVariable)保存文档到数据库之前，把值传递给参数变量。

列表4.1 商品文档的例子

```
{
  _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheelbarrow-9092",
  sku: "9092",
  name: "Extra Large Wheelbarrow",
  description: "Heavy duty wheelbarrow...",
  details: {
    weight: 47,
    weight_units: "lbs",
    model_num: 4039283402,
    manufacturer: "Acme",
    color: "Green"
  },
  total_reviews: 4,
  average_review: 4.5,
  pricing: {
    retail: 589700,
    sale: 489700,
  },
  price_history: [
    {
      retail: 529700,
      sale: 429700,
      start: new Date(2010, 4, 1),
      end: new Date(2010, 4, 8)
    },
    {
      retail: 529700,
      sale: 529700,
      start: new Date(2010, 4, 9),
      end: new Date(2010, 4, 16)
    }
  ],
  primary_category: ObjectId("6a5b1476238d3b4dd5000048"),
  category_ids: [
    ObjectId("6a5b1476238d3b4dd5000048"),
    ObjectId("6a5b1476238d3b4dd5000049")
  ],
  main_cat_id: ObjectId("6a5b1476238d3b4dd5000048"),
  tags: ["tools", "gardening", "soil"],
}
```

① 唯一对象 ID

② 唯一 Slug

③ 嵌套文档

④ 一对多关系

⑤ 多对多关系

文档包含了基本的name、sku、description字段，还有标准的MongoDB object ID字段①。

我们会在下一节里讨论文档的其他方面的问题。

唯一的 URL Slug

此外，我们还定义了 URL slug ^②，wheelbarrow-9092，提供一个有意义的URL结构，利于SEO。MongoDB用户有时候会抱怨URL里无意义的object ID值。

显然，我们不喜欢如下的这种URL结构：

```
http://mygardensite.org/products/4c4b1476238d3b4dd5003981
```

有意义的URL结构是更好的选择：

```
http://mygardensite.org/products/wheelbarrow-9092
```

这种用户友好的永久链接通常叫做slug。我们通常推荐为文档创建一个slug字段来构建有意义的URL。这种字段通常唯一索引，以加速查询和确保唯一。我们也可以在_id里存储slug，用做主键。这个例子我们就不选择它来演示唯一索引了，每个方式都可以接受。假设我们要在products集合里存储商品文档信息，就可以创建如下的唯一索引。

```
db.products.createIndex({slug: 1}, {unique: true})
```

如果slug创建唯一索引，则插入重复值会抛出异常。这时就可以尝试不同的slug值。假设花园商店有多个独轮手推车在销售，那么当销售新的独轮车的时候，就需要创建一个新商品的唯一slug。

下面是执行插入操作的Ruby代码：

```
@products.insert_one({
  :name => "Extra Large Wheelbarrow",
  :sku => "9092",
  :slug => "wheelbarrow-9092"})
```

除非指定，否则驱动会自动确保不会抛出错误。如果插入成功就不会有错，你也可以知道自己插入的是唯一的slug。但是如果抛出异常，那么代码就需要尝试新的slug值。你可以在7.3.2节里查看捕获和优雅地处理异常的方式。

内嵌文档

假设有个key叫details^③，指向一个子文档，它包含商品的详细信息。这个key与_id字段不同，它允许我们可以在一个现有的文档里进行查询。我们可以指定重量、高度单位，还有制造商的型号。我们也可以存储其他的查询字段。例如，如果销售的是种子，可能会包含产量和收获时间；如果卖割草机，可能要包含马力、燃油和保养信息。details字段为这种动态属性数据提供了良好的扩展点。

我们也可以在同一个文档里存储商品的当前和过去的价格。pricing键指向的对象包含零售和批发价格。price_history相反，引用了一个完整的数组价格选项。存储的文档副本很

像版本的控制技巧。

接下来是一个商品tag名字数组。我们可以看到与第1章类似的例子。因为我们可以为数组key建立索引，这是最好和最简单的存储相关标签的方式而且同时可以保证高效的查询效率。

一对多关系

关系怎么处理？通常需要关联其他集合中存储的文档。从把商品和类别关联起来开始❶。首先要定义商品的类别，把商品区分开，假设有单独的类别集合。然后我们需要一个商品和类别的关系❷。这是一对多的关系，因为商品只有一个类别，但是类别可以有多个商品。

多对多关系

我们也想把商品加入它关联的类别里，而不是主要的类别里。这时，商品和类别的关系就是多对多关系。RDMBS中，我们可以使用交叉表来表示多对多的关系。交叉表把多对多的关系存储在单个表里。使用SQL join语句就可以查询与商品相关的所有类别，反之亦然。

MongoDB不支持join连接，所以需要不同的多对多策略。我们定义了一个字段叫做category_ids❸，包含对象ID数组。每个对象ID都作为指向类别文档的指针。

关系结构

列表4.2展示了相同的类别文档。我们可以把它赋值给一个变量，然后调用db.categories.insert(newCategory)方法插入数据库。在以后的查询里可以继续使用而不需要再次输入。

列表4.2 类别文档。

```
{
  _id: ObjectId("6a5b1476238d3b4dd5000048"),
  slug: "gardening-tools",
  name: "Gardening Tools",
  description: "Gardening gadgets galore!",
  parent_id: ObjectId("55804822812cb336b78728f9"),
  ancestors: [
    {
      name: "Home",
      _id: ObjectId("558048f0812cb336b78728fa"),
      slug: "home"
    },
    {
      name: "Outdoors",
      _id: ObjectId("55804822812cb336b78728f9"),
      slug: "outdoors"
    }
  ]
}
```

如果回到商品文档，仔细看下category_ids里的对象ID，就会发现商品关联的是Gardening

Tools类别。商品文档里的category_ids数组键允许我们查询多对多的关系。例如，查询 Gardening Tools类别下的所有商品，代码如下：

```
db.products.find({category_ids: ObjectId('6a5b1476238d3b4dd5000048')})
```

要在商品目录里查询所有的类别，可以使用\$in操作符：

```
db.categories.find({_id: {$in: product['category_ids']}})
```

之前的命令假设product变量已经提前定义了：

```
product = db.products.findOne({"slug": "wheelbarrow-9092"})
```

你会注意到类别文档里的_id、slug、name、description字段。非常简单，但是父文档数组就没有这么简单。为什么要存储如此大量的父文档的类别冗余数据呢？

类别通常都是分级的，而且数据库有多重方式表示它。例如，假设“Home”是商品的类别，“Outdoors”是子类别，“Gardening Tools”又是其子类别。由于MongoDB不支持连接，因此我们违反范式来在每个子文档里存储父类别的名字，这意味着数据是重复存储的。如此，当查询“Gardening Products”类别时，就不需要执行额外的查询来获取父类Outdoors 和 Home 的名字和URL。

有些开发者可能感觉这种级别的去范式设计是无法接受的。但是此刻最好的schema设计就是根据实际的应用程序的设计要求来确定，而并非死背教条理论。当你在下一章里看到更多查询和更新此结构的例子时，许多概念会更加清晰。

4.2.2 用户和订单

如果你关注如何建模用户和订单表，就会看到另外一个公共关系：一对多。这时，每个用户可以有多个订单。在RDBMS数据库中，我们会在orders表中使用外键（这里约定习惯类似）。看看下面的代码列表4.3。

列表4.3 电商订单，包含商品、价格和快递地址

```
{
  _id: ObjectId("6a5b1476238d3b4dd5000048"),
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: "CART",
  line_items: [
    {
      _id: ObjectId("4c4b1476238d3b4dd5003981"),
      sku: "9092",
      name: "Extra Large Wheelbarrow",
      quantity: 1,
      pricing: {
        retail: 5897,
        sale: 4897,
      },
    }
  ],
}
```

非规范化
产品信息

```

    {
      _id: ObjectId("4c4b1476238d3b4dd5003982"),
      sku: "10027",
      name: "Rubberized Work Glove, Black",
      quantity: 2,
      pricing: {
        retail: 1499,
        sale: 1299
      }
    },
    shipping_address: {
      street: "588 5th Street",
      city: "Brooklyn",
      state: "NY",
      zip: 11215
    },
    sub_total: 6196
  }
}

```

← 非规范化销售价总和

第二个订单字段`user_id`存储用户的`_id`。它可以高效地指向购买商品的用户信息，我们会在列表4.4里介绍。

```
db.orders.find({user_id: user['_id']})
```

查询特定订单的用户也非常简单：

```
db.users.findOne({_id: order['user_id']})
```

这种方式使用对象ID作为引用，就可以很方便地在用户和订单之间构建一对多关系。

基于文档思考

我们看一下订单对象的其他方面。通常，订单文档包含商品和传递地址。这些属性，在关系型数据库范式设计模型中分别存储在不同的表中。这里的商品信息使用子文档数组存储，每个子文档表示一个购物车里的商品信息。快递地址只指向一个地址字段的对象。

这种表示有几个优势。首先，是人类思想的胜利。一份完整订单包含商品、快递地址和最终的支付信息，可以封装到单个实体里。当查询数据库时，可以使用单个查询返回整个订单对象。而且，商品交易信息高效地存储在订单文档里。最后，正如将在接下来两章里看到的，你可以方便地执行查询和修改订单文档的操作。

用户文档（列表4.4）展示了相似的模式，因为它也存储了包含`payment_methods`的地址数组文档。正如商品上的`slug`字段，它智能地存储了`username`字段的唯一索引。

列表4.4 带有地址和支付方式的用户文档。

```

{
  _id: ObjectId("4c4b1476238d3b4dd5000001"),
  username: "kbanker",
  email: "kylebanker@gmail.com",
  first_name: "Kyle",

```

```

last_name: "Banker",
hashed_password: "bd1cfa194c3a603e7186780824b04419",
addresses: [
  {
    name: "home",
    street: "588 5th Street",
    city: "Brooklyn",
    state: "NY",
    zip: 11215
  },
  {
    name: "work",
    street: "1 E. 23rd Street",
    city: "New York",
    state: "NY",
    zip: 10010
  }
],
payment_methods: [
  {
    name: "VISA",
    payment_token: "43f6baldfd6b8106dc7"
  }
]
}

```

4.2.3 评价

我们使用商品评价来结束例子数据模型，如列表4.5所示。

每个商品可以有多个评价，而且我们可以通过在评价里存储product_id来实现一对多关系。

列表4.5 商品评价的文档

```

{
  _id: ObjectId("4c4b1476238d3b4dd5000041"),
  product_id: ObjectId("4c4b1476238d3b4dd5003981"),
  date: new Date(2010, 5, 7),
  title: "Amazing",
  text: "Has a squeaky wheel, but still a darn good wheelbarrow.",
  rating: 4,
  user_id: ObjectId("4c4b1476238d3b4dd5000042"),
  username: "dgreenthumb",
  helpful_votes: 3,
  voter_ids: [
    ObjectId("4c4b1476238d3b4dd5000033"),
    ObjectId("7a4f0376238d3b4dd5000003"),
    ObjectId("92c21476238d3b4dd5000032")
  ]
}

```

其余的字段基本是自我描述，易于理解。我们存储了评价的日期、标题和文本；用户提供的评分；用户ID。你可能会好奇为什么要存储username。如果使用RDBMS，就会使用username来关联users表。因为MongoDB不支持join连接，我们可以使用两种方式：根据user集合的每

个评价进行查询或者接受去范式。根据每个评价进行查询也许没有必要，会导致不必要的成本，尤其是当username经常修改的时候。所以，我们这里选择优化查询而不是去范式。

另外值得一提的是，决定在评价文档里保存投票信息。对于用户来说可以选择支持某个评价。这里我们选择在每个评价里保存评论用户的ID。这可以阻止用户多次投票，而且这可以帮助我们查询所有投票的用户。我们缓存了所有有帮助的投票数量，这可以让我们基于投票对于有帮助的评论进行排序。缓存非常有用，因为MongoDB不允许我们查询文档里数组的大小。通过投票来排序评价，例如，投票数组大小缓存在helpful_votes字段里，这是非常有帮助的。

至此，我们已经介绍了基本的电商数据模型。我们也已经看了基本的子文档、数组、一对多和多对多关系，以及如何使用去范式来作为工具简化查询。如果你是第一次看到MongoDB数据模型，理解这个模型则可能需要思想大转换。在接下来的几章里介绍这些问题——从添加唯一投票到修改订单，到智能查询商品。

4.3 核心概念：数据库、集合、文档

Nuts and bolts: On databases, collections, and documents

我们现在从电商网站数据库例子里抽身出来，休息一会，来学习使用MongoDB数据库、集合和文档这些核心概念。这些知识涉及定义、专用特性、极端案例。如果你对于MongoDB分配文件的底层原理，如文档里严格允许的数据类型或者使用封顶集合的优势感兴趣，那就继续阅读下去。

4.3.1 数据库

数据库是集合和索引的命名空间和物理分组。本节里，我们将会讨论创建和删除数据库的细节，深入探讨MongoDB底层如何为单个的数据库分配空间。

管理数据库

MongoDB没有显式的创建数据库的方式。相反，会在第一次写入数据的时候创建数据库。看看下面的Ruby代码：

```
connection = Mongo::Client.new( [ '127.0.0.1:27017' ], :database => 'garden' )
db = connection.database
```

回忆一下，当启动的时候，JavaScript shell执行这个连接，然后允许选择数据库：

```
use garden
```

假设数据库还不存在，在执行上面的代码后还没有创建在磁盘上。那么我们要做的就是创建一个Mongo::DB的实例对象，表示一个MongoDB数据库。只有当我们向集合中写入数据的

时候，才会创建数据库文件。继续Ruby代码，

```
products = db['products']
products.insert_one({:name => "Extra Large Wheelbarrow"})
```

当我们在集合上调用insert_one时，驱动会告诉MongoDB把商品信息插入garden.products集合。如果集合不存在，就创建它。这里还包括在磁盘上创建garden数据库。

通过调用下面的代码来删除集合中的所有数据：

```
products.find({}).delete_many
```

这个代码会删除所有匹配过滤器{}的文档，这就是集合中的所有文档。这个命令不会删除集合，它只会清空集合。要删除集合，就需要使用如下drop方法：

```
products.drop
```

要删除数据库，这意味着要丢弃所有的集合。我们可以通过专门的命令完成。可以使用下面的Ruby代码来删除garden数据库：

```
db.drop
```

在MongoDB shell里，使用JavaScript运行dropDatabase()方法：

```
use garden
db.dropDatabase();
```

删除数据库的时候要格外小心，因为没有办法回滚操作，它会从磁盘上删除数据库文件。

数据库文件和分配

当创建数据库的时候，MongoDB会在磁盘上分配一系列数据库文件集合，包括所有的集合、索引，还有其他元数据。数据库文件存储在启动mongod时dbpath参数指定的目录文件夹里。不指定参数时，mongod会在/data/db^[1]文件夹里存储数据。我们来看看此文件夹下创建garden数据库后的样子：

```
$ cd /data/db
$ ls -lah
drwxr-xr-x 81 pbakkum admin 2.7K Jul 1 10:42 .
drwxr-xr-x 5 root admin 170B Sep 19 2012 ..
-rw----- 1 pbakkum admin 64M Jul 1 10:43 garden.0
-rw----- 1 pbakkum admin 128M Jul 1 10:42 garden.1
-rw----- 1 pbakkum admin 16M Jul 1 10:43 garden.ns
-rwxr-xr-x 1 pbakkum admin 3B Jul 1 08:31 mongod.lock
```

这些文件取决于我们创建的数据库和数据库配置，可能在不同的机器上看起来不一样。首先

^[1]在Windows上，是c:\data\db。如果使用包管理器安装了MongoDB，则有可能在别的地方。例如，在OS X使用Homebrew把数据文件放置到in /usr/local/var/mongodb里。

注意mongod.lock文件，它可以存储服务器进程ID。不要删除或者修改锁文件，除非你正在从宕机事故中恢复数据。如果要启动mongod，就会得到锁文件的错误消息，导致“不干净关机”，而且我们可能必须启动恢复进程。我们会在第11章里深入讨论。

所有的数据库文件都前缀了属于自己的数据库名字。garden.ns是第一个生成的文件。文件扩展名是ns，表示命名空间。数据库中每个集合和索引的元数据有自己的命名空间文件，它的组织形式是哈希表。默认情况下.ns文件固定大小为16MB，它大约允许存储26000个数据。鉴于元数据的大小，这意味着数据库中索引和集合的数目的总和不能超过26000。实际上，不会有这么多索引和集合，但是如果真的需要比这个还多，则我们可以在启动mongod时通过-nssize修改。

处理创建命名空间文件，MongoDB还为集合和索引在文件里分配空间，从0开始以递增证书结束。仔细查看列出的目录，我们可以看到2个核心数据文件，即64 MB garden.0和128 MB garden.1。初始文件的大小可能会吓到新用户。但是MongoDB喜欢以这种预分配空间的方式来确保足够多的数据可以持续存储在文件中。这种方式下，当用户查询和更新数据时，这些操作会在临近的区域执行而不是跨磁盘进行。

随着持续添加数据到数据库，MongoDB会持续分配更多的数据文件。每个新数据文件会是前一个数据文件的两倍大小，直到最大的文件达到2GB的上限为止。从此，后续文件都是2GB。因此，garden.2是256 MB，garden.3是512 MB，以此类推。这里假设数据大小是按照一定的速度增长的，那么数据文件应该持续增长地分配，这是一个常见的分配策略。当然，其后果就是实际分配的空间比使用空间大得多^[1]。

我们可以在JavaScript shell使用stats命令来查询使用的空间和分配的空间：

```
> db.stats()
{
  "db" : "garden",
  "collections" : 3,
  "objects" : 5,
  "avgObjSize" : 49.6,
  "dataSize" : 248,
  "storageSize" : 12288,
  "numExtents" : 3,
  "indexes" : 1,
  "indexSize" : 8176,
  "fileSize" : 201326592,
  "nsSizeMB" : 16,
  "dataFileVersion" : {
    "major" : 4,
    "minor" : 5
  },
  "ok" : 1
}
```

在这个例子里，fileSize字段表示为此数据库分配的总文件空间。这里很简单就是garden

^[1]这在空间有限的环境里会带来问题。此时，应该使用-noprealloc和-smallfiles的组合配置。

数据库2个文件garden.0和garden.1的总空间。dataSize和storageSize的差别很小。前者是数据库里实际BSON数据的大小，后者包括额外的为集合增长预留的空间，还有未删除的空间^[1]。最后，indexSize值展示了数据库索引的总空间。

非常重要的是要注意总索引的大小；数据库性能只有在所有使用的索引都加载到内存里才是最好的。我们将会在第8章和第12章详细解释解决性能问题的技巧。

当我们部署MongoDB时这些都意味着什么？实际上，我们应该使用这些信息来帮助计划需要多少磁盘空间和RAM才能运行MongoDB。我们应该为预期的数据预留足够的磁盘空间，包括一些安全空间。磁盘空间通常很便宜，所以会分配比实际需要多得多的空间。

预估需要的RAM有点困难。我们喜欢有足够的RAM来满足工作数据集合在内存中的需要。工作数据集合（working set）是应用经常访问的数据。在电商网站例子中，可能要访问我们介绍的集合，比如经常在程序运行的时候访问商品和类别集合。这些集合，它们自己的开销和索引的大小应该装入内存，否则就会经常访问磁盘，性能就会受到影响。

这可能是最常见的MongoDB性能问题。也许有其他集合，我们只偶尔需要访问，例如audit审计集合，我们可以把它从工作集合中排除。通常，提前计划足够的内存为正常的应用程序操作集合所用。

4.3.2 集合

集合是结构或概念上相似的文档的容器。这里，我们将更加详细地介绍创建和删除集合，然后介绍MongoDB专门的盖子集合（capped collections）。我们看看例子，展示服务器内核是如何使用集合的。

管理集合

正如我们在前面一节看到的，创建集合是隐式的，只有插入文档时才会创建。但是因为有多重集合类型存在，所以MongoDB也提供了一个创建集合的命令。可以在JavaScript shell里使用：

```
db.createCollection("users")
```

创建标准集合时，可以通过参数指定预分配空间的字节大小。虽然这种做法通常没有必要，但还是可以在JavaScript shell中操作：

```
db.createCollection("users", {size: 20000})
```

集合名字可能包含数字、字母或者圆点字符，但是最好由字母或数字开头。从内部来说，集合名字是通过其命名空间名字来区分的，它包含所属的数据库名字。因此，商品集合技术上

^[1]从技术上来说，集合是在每个数据文件中分配的，这个块叫做扩展(extents)。storageSize是集合扩展分配空间的总空间。

指的是`garden.products`。完全限定的集合名字不能超过128个字符。

圆点符号有时候可以用作虚拟的命名空间。例如，可以使用下面的方式表示一系列集合：

```
products.categories
products.images
products.reviews
```

记住，这只是个组织原则。数据库带原点的集合名字与其他集合是一样的。

集合也可以重命名。例如，可以在shell里使用`renameCollection`方法重命名`products`商品集合：

```
db.products.renameCollection("store_products")
```

盖子集合

除了目前为止创建的标准集合外，还可以创建盖子集合（capped collection，也指有上限的集合，好像盖了盖子一样）。盖子集合最初是为高性能日志场景设计的。它与标准的集合不同，因为有固定的大小。这意味着一旦盖子集合达到最大上限，后续的插入将会覆盖最先插入的文档数据。

当只有最近的数据有价值时，这个设计也避免了用户手动修改集合数据。

要理解如何使用盖子集合，先要假设我们要记录网站用户的行为。这些行为包括查看商品、添加购物车、结账和交易。我们可以编写脚本来模拟日志用户行为到盖子集合里。在这个过程中，我们将会看到一些有意思的集合属性。列表4.6就是例子代码。

列表4.6 模拟日志用户行为到盖子集合里

```
require 'mongo'

VIEW_PRODUCT = 0 # action type constants
ADD_TO_CART = 1
CHECKOUT = 2
PURCHASE = 3

client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'garden')
client[:user_actions].drop
actions = client[:user_actions, :capped => true, :size => 16384]
actions.create

500.times do |n| # loop 500 times, using n as the iterator
  doc = {
    :username => "kbanker",
    :action_code => rand(4), # random value between 0 and 3, inclusive
    :time => Time.now.utc,
    :n => n
  }
  actions.insert_one(doc)
end
```

类型

garden.user_action 集合

例子文档

首先，我们创建一个16KB的集合叫做user_actions，使用的是client创建^[1]。接下来，我们插入500个日志文档^①。每个文档包含一个username、行为代码（0到3的随机数代表）、时间戳，还包括一个自增的整数n，这样我们就可以区分哪个文档过时了。现在我们可以从shell里查询集合了：

```
> use garden
> db.user_actions.count();
160
```

虽然我们插入了500个文档，但只有160个文档保存在集合中^[2]。如果查询集合，就会看到原因了：

```
db.user_actions.find().pretty();
{
  "_id" : ObjectId("51d1c69878b10e1a0e000040"),
  "username" : "kbanker",
  "action_code" : 3,
  "time" : ISODate("2013-07-01T18:12:40.443Z"),
  "n" : 340
}
{
  "_id" : ObjectId("51d1c69878b10e1a0e000041"),
  "username" : "kbanker",
  "action_code" : 2,
  "time" : ISODate("2013-07-01T18:12:40.444Z"),
  "n" : 341
}
{
  "_id" : ObjectId("51d1c69878b10e1a0e000042"),
  "username" : "kbanker",
  "action_code" : 2,
  "time" : ISODate("2013-07-01T18:12:40.445Z"),
  "n" : 342
}
```

文档按照插入的顺序返回。查看n的值可知，集合中最老的值是n等于340，这意味着从0~339已经老化了。因为这个盖子集合最大是16 384 kB，只能包含160个文档，每个文档大概102B。我们会在下一节里确认这个假设。尝试去为文档添加一个字段以增加平均的文档大小，从而减少总的文档数量。

除了大小限制以外，MongoDB允许为盖子集合指定最大文档数量的max参数。这允许对存储文档的总数进行细粒度控制。记住，大小配置具备优先权。创建集合的方式如下所示：

```
> db.createCollection("users.actions",
  {capped: true, size: 16384, max: 100})
```

盖子集合不允许正常集合的所有操作。例如，不允许从盖子集合里删除单个文档，而且也不

^[1] 等价的创建命令可以从shell里使用db.createCollection("user_actions",{capped: true, size: 16384})。

^[2] 数字可能不同，这和MongoDB版本有关系；值得注意的部分就是少于这个数字的文档被插入集合中了。

能增加文档的大小。盖子集合最初是为日志设计的，所以没有必要实现删除和更新文档操作。

TTL 集合

MongoDB 也允许在特定的时间后废弃文档数据，有时候叫做生存时间 time-to-live (TTL) 集合，这个功能实际上是通过一个特殊的索引实现的。创建 TTL 索引的方式如下：

```
> db.reviews.createIndex({time_field: 1}, {expireAfterSeconds: 3600})
```

这个命令会在 (time_field) 字段上创建索引。这个字段会定期检查时间戳，与当前的时间值比较。如果 time_field 与当前时间值的差距大于 expireAfterSeconds 设置，文档就会自动被删除。这个例子里，评价文档会在一个小时后删除。

使用 TTL 索引，假设我们已经在 time_field 里存储了时间戳。

以下是存储时间戳的例子代码：

```
> db.reviews.insert({
  time_field: new Date(),
  ...
})
```

这个代码在插入时为 time_field 设置时间值。我们也可以插入其他的时间值，比如将来的时刻。记住，TTL 索引只会比较索引值和当前值的差别，比较当前的时间间隔与 expireAfterSeconds 值的大小。因此，如果我们在字段里设置将来的值，那么它只会等到未来的某个时刻超过了设定的 expireAfterSeconds 的长度。这个功能可以管理文档的生命周期。

TTL 索引还有几个限制。我们不能在 _id 字段建立 TTL 索引，或者在其他已经建立索引的字段建立 TTL 索引。我们也不能在盖子集合里使用 TTL 索引，因为它不支持删除单个文档。最后，虽然我们可以在索引字段里有一组时间戳，但也不能组合 TTL 索引。这种情况下，TTL 属性只会使用集合中最早的时间戳。

实际上，你可能发现自己不会要用到 TTL 集合，其实它在某些常见下是非常有用的，所以最好记住它。

系统集合

MongoDB 的部分设计依赖于内部集合的使用。有两个特殊的集合：system.namespaces 和 system.indexes。我们可以在前者里查询当前数据库定义的所有命名空间：

```
> db.system.namespaces.find();
{ "name" : "garden.system.indexes" }
{ "name" : "garden.products.$_id_" }
{ "name" : "garden.products" }
{ "name" : "garden.user_actions.$_id_" }
{ "name" : "garden.user_actions", "options" : { "create" : "user_actions",
" capped" : true, "size" : 1024 } }
```


system.indexes存储了当前数据库里定义的所有索引。为了查看当前garden数据库定义的索引，可以插入这个集合：

```
> db.system.indexes.find();
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "garden.products", "name" : "_id_" }
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "garden.user_actions", "name" :
"_id_" }
{ "v" : 1, "key" : { "time_field" : 1 }, "name" : "time_field_1", "ns" :
"garden.reviews", "expireAfterSeconds" : 3600 }
```

system.namespaces和system.indexes都是标准的集合，调试的时候非常有用。MongoDB为了复制功能也使用了盖子集合，复制功能可以保证多个mongod数据库之间的数据同步。每个主从复制集在特殊的盖子集合oplog.rs里记录了所有的写操作日志。从节点从这个集合里依次读取数据，应用到自己的数据库中。我们会在第10章里详细讨论这个问题。

4.3.3 文档和插入

我们将在本章里详细介绍文档和插入的细节内容。

文档序列化、类型和限制

所有的文档在发送给MongoDB之前都序列化为BSON格式，以后再从BSON反序列化。驱动库会处理底层的数据类型转换工作。绝大部分驱动都提供了从BSON序列化的简单接口，当读/写文档的时候会自动完成，我们并不需要担心这个过程。这里的例子仅仅是用于演示。

在之前的盖子集合里，可以假设文档大约是102B。我们也可以通过Ruby 驱动的BSON serializer序列化器来验证这个假设：

```
doc = {
  :_id => BSON::ObjectId.new,
  :username => "kbanker",
  :action_code => rand(5),
  :time => Time.now.utc,
  :n => 1
}
bson = doc.to_bson
puts "Document #{doc.inspect} takes up #{bson.length} bytes as BSON"
```

这个序列化方法会返回一个字节数组。如果运行这个代码，就会获取一个82B的对象，它和预估的差距不大。82B和102B的差别在于正常的集合和文档的其他开销。MongoDB为集合分配了特定的空间，一定要用于存储元数据。此外，在正常的集合（非盖子集合）里，更新文档可能导致空间增长，需要移动新的地址，并且留出空白区^[1]。这个特性导致数据大小和MongoDB占用的磁盘空间之间有差别。

^[1]更多细节可以查看配置指令的填充因子。填充因子确保为文档增长预留一定的空间。填充因子从1开始，所以，第一次插入时没有额外分配空间。

使用StringIO帮助类反序列化BSON非常简单。尝试运行Ruby代码来验证它的工作：

```
string_io = StringIO.new(bson)
deserialized_doc = String.from_bson(string_io)
puts "Here's our document deserialized from BSON:"
puts deserialized_doc.inspect
```

注意，不能序列化任意哈希数据结构。要做到无错序列化，key名字就必须有效，而且每个值必须可以转换为BSON类型。有效的key名字由最大255B长度的字符串组成。字符串可以由任意合法的ASCII字符组成，但有3个例外：不能由\$开始、不能包含圆点、除了在最后以外不能包含null字节。当使用Ruby编程时，可以使用符号作为哈希key，但是它们也会在序列化时转换为等价的字符串。

这看起来有些奇怪，其实是因为key名也会存储在文档里，会影响数据的大小。这与RDBMS关系型数据库相反，它们的列名与行名是分离的。当使用BSON时，如果使用dob代替date_of_birth，则每个文档就可以节约10B的空间。看起来不多，但一旦有10亿个文档，就可以节约10GB的空间。当然也不是说尽量使用不合理、短小的key名字，要看情况而定。但是如果有大量的数据，那么合适的key名字确实是可以节约空间的。

除了有效的key名字外，文档也必须包含可以序列化为BSON的值。我们可以在<http://bsonspec.org>查看BSON类型的表，带有例子和注释。我们这里只介绍一些重要的数据类型。

字符串

所有的字符串必须使用UTF-8编码。虽然UTF-8已经很快成为字符编码的标准，但是还有许多时候在使用旧的编码方式。用户在使用旧的编码导入数据到MongoDB时会遇到问题。解决办法通常是先转换为UTF-8，再进行插入，或者直接把文本存储为BSON二进制类型^[1]。

数字

BSON指定了三种数据类型：double、int、long。这意味着BSON可以编码任意IEEE浮点值以及任意8B长度的有符号整数。在动态语言比如Ruby和Python里序列化整数时，驱动会自动确定是否编码为int或者long int。事实上，只有一种情况需要显示指定类型：当我们通过Javascript shell插入数据类型时。Javascript，不幸的是，只支持一种数据类型叫做Number，它等价于IEEE 754 Double。所以，如果希望从shell保存一种数据类型，作为整数形式，我们需要显示指定类型，使用NumberLong()或NumberInt()。看看以下例子：

```
db.numbers.save({n: 5});
db.numbers.save({n: NumberLong(5)});
```

我们已经为numbers集合保存了2个文档，虽然它们的值是相等的：第一个以double保存，第二个以long int保存。查询n为5的文档，会返回所有的数据：

^[1]如果你不熟悉字符编码，可以去阅读 Joel Spolsky 著名的文章(<http://mng.bz/LVO6>)。

```
> db.numbers.find({n: 5});
{ "_id" : ObjectId("4c581c98d5bb5eb2365a838f9"), "n" : 5 }
{ "_id" : ObjectId("4c581c9bd5bb5eb2365a838fa"), "n" : NumberLong( 5 ) }
```

我们可以看到第2个值标记为long int。另外一种方式是使用专门的\$type操作符查询BSON类型。每个BSON类型通过一个从1开始的整数标识。如果查看BSON的文档<http://bsonspec.org>，会看到double是类型1，而64b的整数类型是18。因此，可以通过类型来查询集合：

```
> db.numbers.find({n: {$type: 1}});
{ "_id" : ObjectId("4c581c98d5bb5eb2365a838f9"), "n" : 5 }
> db.numbers.find({n: {$type: 18}});
{ "_id" : ObjectId("4c581c9bd5bb5eb2365a838fa"), "n" : NumberLong( 5 ) }
```

这个验证了存储数据的差别。我们可能从来不会在生产环境里使用\$type操作符，但是正如这里看到的，它确实是很棒的调试工具。

BSON数值类型最常见的问题就是缺少小数位支持。这意味着如果我们计划在MongoDB里存储货币数据值，就需要使用整数类型，并且单位是分。

时间

BSON的实际类型用来存储临时值。从Unix纪元计时开始时间值使用64b整数的毫秒值表示，负数表示之前的时间^[1]。

使用要注意以下几点。首先，如果要在JavaScript里创建日期，记住JavaScript的月份是从0开始的。这意味着new Date(2011,5,11)表示2011年6月11日。其次，如果使用Ruby驱动存储类似数据，BSON序列化器会期望一个UTC格式的Ruby Time。

因此，我们不能使用日期类型表示时间区间，因为BSON时间不能编码此时间数据。

虚拟类型

如果非要用特定的时区来存储时间，怎么办？基本的BSON类型不能满足需求时，虽然没有创建自定义BSON类型的方法，但可以使用不同的基元BSON类型在子文档里创建自定义虚拟类型。例如，如果要存储带有时区的时间，则可以使用如下的文档结构。Ruby代码如下：

```
{
  time_with_zone: {
    time: new Date(),
    zone: "EST"
  }
}
```

编写处理这种组合文档类型的应用代码其实不难，这也是实际项目中处理问题的方式。例如，用Mongo-Mapper、Ruby编写的MongoDB对象映射器允许我们通过定义to_mongo和from_mongo方法来处理自定义数据类型。

^[1] UNIX 纪元从 1970 年 1 月 1 日午夜开始计算，UTC 时间。我们会在 3.2.1 节简要介绍这个概念。

文档的限制

在MongoDB 2.0和以后的版本里，BSON文档大小限制为16MB^[1]。限制它的原因有2个。第一，阻止开发者创建无意义的数据库模型。虽然差的数据模型仍然可能出现，但16MB的限制不鼓励使用超过这个限制的文档。如果要存储超过16MB的文档，就可以考虑把文档存储为更小的文档。或者考虑MongoDB文档是否是一个合适的存储库——或许它更适合管理小文件。

第二，16MB的限制与性能相关。在服务器段，查询大的文档时需要在发送给客户端之前把文档拷贝到缓存里。这个拷贝工作非常昂贵，特别是当客户端不需要整个文档时^[2]。此外，一旦发送数据，这就是跨网络传输数据的工作，而且客户端驱动要反序列化它，成本很高，尤其是处理上吉字节的大文件数据时。

MongoDB文档的嵌套深度最大值限制是100。嵌套就是文档里包含新的文档。使用深嵌套文档——例如，如果要序列化一个树形结构到MongoDB文档中——其结果就是查询和访问非常困难，还可能会导致其他问题。这种类型的数据结构通常都通过递归函数访问，对深入嵌套的文档处理会导致堆栈溢出。

如果你遇到了文档大小和嵌套的限制，最好把它们分割开，修改数据模型，或者使用其他的集合进行存储。如果要存储大的二进制对象，比如图片或者视频，这就是完全不同的情况。请参考附录C对于大二进制对象的处理技术。

大量插入

只要验证问答有效，插入过程就十分简单。关于插入文档最有价值的细节，包括对象ID生成、如何在网络层实现以及检查异常，在第3章里都做了介绍。但是一个重要的特性——大量插入，仍然值得在这里详细讨论。

所有的驱动几乎都可以一次插入多个文档。这对于插入大量数据非常方便，尤其是从另外一个数据库里导入大量数据时。这是一个简单的Ruby大量插入的例子：

```
docs = [ # define an array of documents
  { :username => 'kbanker' },
  { :username => 'pbakkum' },
  { :username => 'sverch' }
]
@col = @db['test_bulk_insert']
@ids = @col.insert_many(docs) # pass the entire array to insert
puts "Here are the ids from the bulk insert: #{@ids.inspect}"
```

大量插入返回对象的ID数组，而不是单个ID。这是MongoDB驱动的标准。

^[1]不同版本的限制可能不同。要查询当前服务器版本的限制可以在当前 shell 里运行 `db.isMaster()`，查询 `maxBsonObjectSize` 值。如果没有找到这个字段，限制就是 4MB（使用的是很老版本的 MongoDB）。我们可以在这里看到更多的信息 <http://docs.mongodb.org/manual/reference/limits>。

^[2]正如将会在下一章看到的，我们可以指定文档的某个字段在返回查询里来限制应答的大小。如果经常这么做，可以考虑重新评估数据模型。

大量插入对于性能非常有帮助。这意味着只需要一个连接就可以插入大量数据，而不需要大量独立地来回往返。这个方法有个限制，所以，如果要插入100万个文档，我们就必须把它们分割为多个大量插入的文档组^[1]。

用户经常询问到底多大的插入是理想的，答案取决于许多因素。理想的数字可能从10到200。这里实践是检验真理的唯一标准。数据库设置的插入操作上限是16MB。经验证明，最有效的大量插入在这个限制下都可以正常工作。

4.4 总结

Summary

本章介绍了许多深入的内容，祝贺你坚持学习到这个深度！

我们从schema设计的理论讨论开始，然后实战设计了电商网站的数据模型。这给了我们一个机会来看看生产环境下的文档是什么样子，这应该激发我们去思考传统关系型数据库RDBMS和MongoDB之间的schema更具体的差别。

本章结束部分我们又深入了解了数据库、文档和集合的知识；我们可能后面会继续复习，把本节作为参考知识。我们已经解释了MongoDB的雏形，但是还没有开始移动数据。下一章里会继续深入学习ad hoc查询的强大威力。

^[1] 批量插入的限制是16MB。

构建查询

Constructing queries

本章内容

- 查询一种电子商务数据模型
- MongoDB查询语句的详细信息
- 查询选择器及其选项

MongoDB不使用SQL，它使用自己的JSON查询语言。通过这本书，我们已经对这门语言进行了探索，现在就转向一些耐人寻味的真实存在的例子。我们将会重新引入前面的章节介绍的商务数据模型和目前各种针对它的查询。这些查询包括_id查找、限定范围、排序和投影。

记住，在这一章中学习的MongoDB查询语句和聚合函数（在第6章中介绍）仍然在进程中运行，并在每个新加入的版本中改进。但在MongoDB中掌握查询和聚合并不能映射出每一个角落，因为它总是寻找最佳的方式来完成日常任务。通过本章中的示例，我们采取最清晰的学习线路。通过本章的学习，就会对MongoDB的查询有了直观的了解，准备好把这些工具应用到应用程序设计中。

5.1 电子商务查询

E-commerce queries

这一节我们继续探索在上一章中提到的商务数据模型。我们已经定义的文档结构包含了商品、类别、用户、订单和商品评论。现在，铭记结构，我们将展示在一个典型的电子商务应用中查询这些实体。当然，有一些查询比较简单。例如，_id查询在这个点上不会是神秘莫测的。但是我们会展示一些复杂模式，包括查询为显示类别的层次结构以及提供的产品清单视图。此外，我们应通过为这些查询寻找一些可能的索引来保持效率。

5.1.1 产品、类别和评论

大多数电子商务应用包含产品和类别这两个基本的视图。第一个是产品的首页，突出显示给

出的商品、展示评论和给出某种意义上的产品类别。第二个是商品的表单页，允许用户查看类别层次结构和所选商品类别的缩略视图。让我们从商品的首页开始，在很多方法中选择两种简单的。

设想一下我们商品页面的URL使用了slug关键字（我们在第4章学习的友好链接）。这种情况下，我们使用下面3条查询语句来获取商品页面需要的所有数据。

```
product = db.products.findOne({'slug': 'wheel-barrow-9092'})
db.categories.findOne({'_id': product['main_cat_id']})
db.reviews.find({'product_id': product['_id']})
```

第一条查询语句查找商品的slug为 wheel-barrow-9092。当我们有了商品时，就可以在类别集合使用简单的_id查询语句来查询类别信息。最后，可以使用另外一种简单的查询语句查找关于商品的所有评论。

FINDONE 与 FIND QUERIES

我们发现前两种使用findOne方法，后面使用find方法代替了。所有的MongoDB驱动提供了这两种方法，因此这两种方法之间的区别值得探讨。比如在第3章中所介绍的，find方法返回一个光标（对象），而findOne方法返回一个文件。findOne方法类似下面这样，当使用应用限制时返回光标：

```
db.products.find({'slug': 'wheel-barrow-9092'}).limit(1)
```

如果想得到单一文件，则当文件存在时findOne方法会返回文件。如果我们需要返回多个文档，则使用find方法。我们需要在程序的某些地方使用遍历光标。

如果我们使用findOne在数据库中查询匹配多个项目，它就会在自然排序文件集合中返回第一个项目。大多数情况下（并不总是如此）文档以相同的顺序插入集合与集合上限中，并且总是如此。如果希望得到多个文档结果，就尽可能使用find查询并显式对结果进行排序。

现在再次查看商品页面的查询，是否有任何不妥？如果查询对于评论看起来比较宽松，就是对的。这种查询可以返回指定商品的所有评论，但是当这种商品有数百条评论的时候这样做可能不太严谨。

忽略，限制和排序查询选项

大多数的应用程序分页评论，为此MongoDB提供了忽略和限制选项。我们可以使用这些选项进行分页评论文件：

```
db.reviews.find({'product_id': product['_id']}).skip(0).limit(12)
```

注意，我们通过在find的返回结果集上调用skip与limit方法来设置这些参数。这与我们通常，甚至其他MongoDB驱动中看到的模式不同，所以可能会造成困惑。它们在查询被调用后出现，而排序和限制参数发送至查询并由MongoDB服务器处理。这种语法模式被称为方法链接，旨

在更容易地生成查询语句。

如果想要排序一致地显示评论，那么意味着要对查询结果进行排序。如果要对每个评论所收到的有益投票进行排序，则可以简单地这样指定：

```
db.reviews.find({'product_id': product['_id']}).
    sort({'helpful_votes': -1}).
    limit(12)
```

总之，这个查询告诉MongoDB返回排序在总评论中最有益投票前12条，并按降序排列。现在，有了忽略、限制和排序存在的位置，需要决定是否把分页放在第一位。为此，可以发出一个计数查询。使用计数查询结果和页面评论相结合就得到我们需要的结果，为商品页面做的查询就完成了：

```
page_number = 1
product = db.products.findOne({'slug': 'wheel-barrow-9092'})
category = db.categories.findOne({'_id': product['main_cat_id']})
reviews_count = db.reviews.count({'product_id': product['_id']})
reviews = db.reviews.find({'product_id': product['_id']}).
    skip((page_number - 1) * 12).
    limit(12).
    sort({'helpful_votes': -1})
```

调用skip、limit和sort的顺序在JavaScript shell中并不重要。

这些查找可以使用索引。标签拥有唯一索引，因为它们提供了备用主键，而所有的_id字段自动获取标准合集的唯一索引。这种情况下，评论集合需要包含user_id和product_id字段。

商品列表页面

商品首页面查询到位，现在可以转到商品列表页面。这个页面展示选定类别可浏览的商品列表。商品总类和分类的链接也将展示在页面上。商品列表页面定义自身的类别，因此，页面请求将包含类别的标签：

```
page_number = 1
category = db.categories.findOne({'slug': 'gardening-tools'})
siblings = db.categories.find({'parent_id': category['_id']})
products = db.products.find({'category_id': category['_id']}).
    .skip((page_number - 1) * 12)
    .limit(12)
    .sort({'helpful_votes': -1})
```

任何类别不具有相同的总类ID，因此分类查询简单明了。因为商品包含了类别ID数组，所以在一个给定的类别中查找所有商品是很容易的。我们还可以想出其他替代的排序方法（比如名字，价格等等）。对于这种情况，我们可以更改排序字段。

考虑这些排序是否有效很重要。我们可以依靠索引来处理排序，但是当添加了更多的排序选项时，随着索引数目的增加，维护索引可能变得不太合理，因为每个索引的写入代价都会比

较高。我们将在第8章中进一步讨论，但是这里要思考这些以权衡利弊。

商品列表页面有个默认显示，当访问根级目录时，不会显示任何商品。针对查询的类别集合的父ID为null的查询，搜索结果是根级别类别：

```
categories = db.categories.find({'parent_id': null})
```

5.1.2 用户和订单

上一节中的查询仅限于_id查找和排序。再看用户和订单，就要深入更多，我们想得到订单的基本报告。事例查询搜索文件看起来像第4章的列表4.1（商品）和4.4（用户）。

让我们从简单的用户登录验证开始，就是提供用户名和用户密码让用户登录的应用。因此，我们希望经常看到下面的查询：

```
db.users.findOne({
  'username': 'kbanker',
  'hashed_password': 'bd1cfa194c3a603e7186780824b04419'})
```

如果用户名存在并且密码正确，就会得到整个用户文档；否则，不返回任何东西。这种查询虽然是可以接受的，但我们只想检查用户名是否存在，能否不用返回整个用户文件。我们可以投影返回限制字段。

```
db.users.findOne({
  'username': 'kbanker',
  'hashed_password': 'bd1cfa194c3a603e7186780824b04419'},
  {'_id': 1})
```

在JavaScript shell中投影就需要传递一个额外的参数：想得到的一个值为1的哈希字段。我们将在第5.2.2节中深入探讨投影。如果已经熟悉SQL和RDBMS，这就相当于是SELECT*和SELECT ID的不同。响应包含了完整的_id字段文档：

```
{ "_id": ObjectId("4c4b1476238d3b4dd5000001") }
```

用户部分匹配查询

我们可能想要用其他方式查询用户集合，比如搜索名称。通常希望用单个字段进行查找，如last_name：

```
db.users.find({'last_name': 'Banker'})
```

这种方法很有效，但是有完全匹配的限制搜索。其一，你可以不知道如何拼写给定的用户名。这种情况下，可以使用部分匹配查询方式。假设我们知道用户的姓由字母Ba开头。MongoDB允许使用正则表达式查询：

```
db.users.find({'last_name': /^Ba/})
```

正则表达式/^Ba/可以读作“行的开头是字母B，紧接着是字母a”。像这样的前缀搜索可以使

用索引，但并不是所有的正则表达式都可以使用索引。

查询特定范围

当对用户进行营销时，最想要的是特定范围内的用户。例如，想查询居住在曼哈顿的用户时，就查询这个范围内用户的邮政编码。

```
db.users.find({'addresses.zip': {'$gt': 10019, '$lt': 10040}})
```

回想一下，每个用户文件包含一个或多个地址文档。这种查询将匹配用户文档，如果这些地址和邮政编码属于给定范围，就可以使用\$gte（大于）和\$lt（小于）运算符来定义范围。为使这种查询有效，就要在addresses.zip中定义索引。

在下一章中，我们会看到更多查询数据的例子。接下来，我们学习使用MongoDB聚合函数对数据进行了解。在掌握了这些知识后，就深入看看MongoDB查询语言，特别解释一下通用的语法和操作符。

5.2 MongoDB 的查询语言

MongoDB's query language

现在来探索MongoDB查询语言的荣耀。我们已经学习了一些真实的查询实例，本节旨在为MongoDB查询功能进行更全面的介绍。如果是第一次学习关于MongoDB查询的知识，就可能更容易理解本节内容。当我们为程序编写更高级查询时，还要重新复习本节内容。

5.2.1 查询条件和选择器

查询条件允许我们使用单个或者多个查询选择器来指定查询结果。MongoDB提供多种可能的选择器。本节进行概述。

选择器匹配

指定查询最简单的一种方式是使用查询器的键值对字面匹配我们要查找的文件。下面是几个例子：

```
db.users.find({'last_name': "Banker"})
db.users.find({'first_name': "Smith", birth_year: 1975})
```

第二种查询是读取，如查出first_name为史密斯且出生在1975年的所有用户。注意，传递多个键值对时两者必须匹配；查询条件函数可以为布尔值。如果想要展示一个布尔值，就参阅下面的布尔值运算符章节。

在MongoDB中所有的文本字符串匹配都是区分大小写的。如果需要不区分大小写匹配，可使

用正则表达式项（在本章的后面解释，我们将使用*i*正则表达式标识）或者探讨使用第9章介绍的文本搜索。

范围

我们可能经常需要查询值在一定跨度范围内的文档。在大部分语言中，使用<、<=、> 和 >=。在MongoDB中，我们会得到类似的一组运算符\$lt、\$lte、\$gt和\$gte。如我们期望的那样，我们在本书中会经常使用这种运算符。表5.1展示了最常使用的范围查询运算符。

表 5.1 范围查询运算符摘要

运算符	描述
\$lt	小于
\$gt	大于
\$lte	小于等于
\$gte	大于等于

初学者经常努力组合这些运算符。一个常见的错误就是重复搜索键：

```
db.users.find({'birth_year': {'$gte': 1985}, 'birth_year': {'$lte': 2015}})
```

上述查询只考虑了最后一个条件。我们可以按照下面的方式查询：

```
db.users.find({'birth_year': {'$gte': 1985, '$lte': 2015}})
```

我们也应该了解它们在不同的数据类型之间如何工作。范围查询匹配值仅仅当它们含有相同类型的值时才进行比较。例如，假设有如下文件集合：

```
{ "_id" : ObjectId("4caf82011b0978483ea29ada"), "value" : 97 }
{ "_id" : ObjectId("4caf82031b0978483ea29adb"), "value" : 98 }
{ "_id" : ObjectId("4caf82051b0978483ea29adc"), "value" : 99 }
{ "_id" : ObjectId("4caf820d1b0978483ea29ade"), "value" : "a" }
{ "_id" : ObjectId("4caf820f1b0978483ea29adf"), "value" : "b" }
{ "_id" : ObjectId("4caf82101b0978483ea29ae0"), "value" : "c" }
```

就可以使用下面的查询：

```
db.items.find({'value': {'$gte': 97}})
```

你可能会认为查询应该返回6个文件，因为字符串在数值上等值于整数97、98和99。但这种情况并非如此。由于MongoDB是无schema的，查询返回整数仅仅是因为提供的标准本身是一个整数。如果想得到字符串结果，就必须使用字符串查询：

```
db.items.find({'value': {'$gte': "a"}})
```

我们不需要担心这种类型的限制，只要不存储多种类型为同一集合的密钥即可。这是一种非常好的通用实践，我们应该遵守它。

设置运算符

三种查询运算符——\$in、\$all和\$nin——把一个或多个值的列表作为它们的谓语，因此被称为集合运算符。如果给定的值匹配搜索关键字，\$in返回文档。我们可以使用这些运算符返回分立的类别集合的所有商品。表5.2展示了这些设置查询运算符。

表 5.2 集合操作符总结

操作符	描述
\$in	如果任意参数在引用集合里，则匹配
\$all	如果所有参数在引用集合里且被使用在包含数组的文档中，则匹配
\$nin	如果没有参数在引用的集合里，则匹配

如果有下面这些类别IDs列表

```
[
  ObjectId("6a5b1476238d3b4dd5000048"),
  ObjectId("6a5b1476238d3b4dd5000051"),
  ObjectId("6a5b1476238d3b4dd5000057")
]
```

对应割草机、工具和工作服这些类别，则查询找到相应类别的所有商品：

```
db.products.find({
  'main_cat_id': {
    '$in': [
      ObjectId("6a5b1476238d3b4dd5000048"),
      ObjectId("6a5b1476238d3b4dd5000051"),
      ObjectId("6a5b1476238d3b4dd5000057")
    ]
  }
})
```

使用\$in运算符的另一种方法是作为布尔的包容性或者对单一属性。用这种表达方式，可能会读取前面的查询：“找出类别是割草机或手工具或工作服的所有商品。”注意，如果需要使用布尔值OR的多种属性，就需要使用\$or运算符，这在下一节中叙述。

■ \$in 频繁使用IDs列表

■ \$nin (not in) 仅返回一个没有给定元素匹配项的文档。我们可以使用\$nin查找所有商品，既不是黑也不是蓝：

```
db.products.find({'details.color': {'$nin': ["black", "blue"]}})
```

■ \$all 如果每个给定元素匹配搜索关键字，则匹配。如果想找到所有商品标签，如礼品和花园，则\$all是一个很好的选择：

```
db.products.find({'tags': {'$all': ["gift", "garden"]}})
```

当然，这种查询的意义仅仅是用标签属性存储一组数组项，比如这样：

```
{
  'name': "Bird Feeder",
```

```
'tags': [ "gift", "birds", "garden" ]
}
```

选择性是一种使用索引缩小查询结果的能力。由于\$ne 和\$nin运算符不是选择的，因此，当使用集合运算符时，记住\$in 和\$all可以利用索引，但\$nin不能因此而要集合浏览。如果我们使用\$nin，就尝试使用它结合另一个使用索引的查询项。最好找到不同的方式来表示这种查询。

布尔运算符

MongoDB 布尔运算符包括\$ne、\$not、\$or、\$and、\$nor 和 \$exists。表5.3总结了布尔运算符。

表 5.3 布尔运算符总结

操作符	描述
\$ne	不匹配参数条件
\$not	不匹配结果
\$or	有一个条件匹配就成立
\$nor	所有条件都不匹配
\$and	所有条件都匹配
\$exists	判断元素是否存在

\$ne，并不等同于运算符，作用并不如我们所预期。在操作中，最佳使用方式是结合一个其他的运算符；否则，会因为不是使用索引而造成效率底下。比如，可以使用\$ne查找Acme制造的所有没有gardening标签的商品：

```
db.products.find({'details.manufacturer': 'Acme', tags: {$ne: "gardening"}})
```

\$ne适用于关键字指向的单一值或者数组，如示例中展示匹配的标签数组。

\$ne匹配指定值，\$not不匹配MongoDB的运算符或者正则表达式的查询结果。大多数查询操作已经否定匹配形式（\$in和\$nin，\$gt和\$lte等）；\$not非常有用，因为它可以过滤不符合表达式条件的结果。参考下面的例子：

```
db.users.find({'age': {'$not': {'$lte': 30}}})
```

如我们预期的那样，这种查询返回的文档是age大于30。当然也返回没有age字段的文档，这种情况下使他区别于使用\$gt运算符。

\$or表示使用两个不同的关键字逻辑分离两个值。有个重点：如果可能值的范围是相同的關鍵字，就使用\$in替代。查找所有商品是蓝色或者绿色，代码就像这样：

```
db.products.find({'details.color': {$in: ['blue', 'Green']}})
```

要找到是蓝色或者由Acme生产的所有商品，就用到\$or：


```
db.products.find({
  '$or': [
    {'details.color': 'blue'},
    {'details.manufacturer': 'Acme'}
  ]
})
```

\$or采用数组查询选择器，每个选择器可以是任意复杂和可能自身包含其他查询运算符。\$nor的工作原理和\$or的大致相同，但仅当没有其查询选择器为真时逻辑为真。

像\$or一样，\$and运算符也采用一组数组查询选择器。因为MongoDB诠释所有查询选择器包含更多关键字使用ADNing的条件，我们仅当不能使用更简单的方式表示AND关系时才使用\$and运算符。例如，假如我们要查找标签包含礼品或者节日且无论是园艺或者绿化的所有商品。表示该查询的方法只有结合两个\$in查询语句：

```
db.products.find({
  $and: [
    {
      tags: {$in: ['gift', 'holiday']}
    },
    {
      tags: {$in: ['gardening', 'landscaping']}
    }
  ]
})
```

查询关键字文档

这一节我们学习最后一个运算符\$exists。这个运算符是必须的，因为集合不会强制执行一个固定的模式，所以我们偶尔需要一种方式来查询包含特定关键字的文档。记得我们计划使用商品details属性来存储自定义字段。例如，可能我们存储一个颜色字段在details属性中。但是如果一个子集的所有商品指定一种颜色，我们就可以查询那些不同的属性，像这样：

```
db.products.find({'details.color': {$exists: false}})
```

也可能需要做相反的查询：

```
db.products.find({'details.color': {$exists: true}})
```

这里我们可以检查文档中是否存在该字段。甚至即使字段存在，也还是可以设置为null。根据我们的数据和查询，可以很好地筛选这些值。

匹配子文档

本书中的一些商务数据模型实体拥有指向单一嵌入对象的关键字。商品的details属性是个很好的例子。这是相关文档的一部分，表示为JSON：

```
{
  _id: ObjectId("4c4b1476233d3b4dd5003981"),
  slug: "wheel-barrow-9092",
```

```
sku: "9092",
details: {
  model_num: 4039283402,
  manufacturer: "Acme",
  manufacturer_id: 432,
  color: "Green"
}
```

我们使用一个点(.)分离有关关键字查询此类对象。例如, 如果想找到Acme公司生产的所有商品, 就可以这样查询:

```
db.products.find({'details.manufacturer': "Acme"});
```

这种查询可以指定任意深度, 例如做了修改后的表示形式如下:

```
{
  _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",
  details: {
    model_num: 4039283402,
    manufacturer: {
      name: "Acme",
      id: 432
    },
    color: "Green"
  }
}
```

查询选择器中的关键字包含两个点(.):

```
db.products.find({'details.manufacturer.id': 432});
```

除了匹配单一子文档属性, 还可以匹配整个项目。例如, 假设我们正在使用MongoDB存储股票市场的位置, 那么为了节约空间, 就放弃使用标准对象的ID而使用由股票符号和时间戳组成的复合关键字。下面是一个代表文档:

```
{
  _id: {
    sym: 'GOOG',
    date: 20101005
  },
  open: 40.23,
  high: 45.50,
  low: 38.81,
  close: 41.22
}
```

然后使用_ID查询, 就可以找到GOOG的2010年10月2号的总结:

```
db.ticks.find({'_id': {'sym': 'GOOG', 'date': 20101005}})
```

重点是要意识到像这样查询匹配一个实体对象, 将执行严格的逐字节比较, 关键字的顺序非

常重要。下面的查询不等同也不会匹配相同的文档：

```
db.ticks.find({'_id': {'date': 20101005, 'sym': 'GOOG'}})
```

关键字的顺序通过命令在JSON文档中保存，这不一定适用于其他语言，且更安全的方式是假设不会保留顺序。例如，哈希在Ruby1.8不会保留顺序。在Ruby1.8中保留关键字顺序，必须使用类的对象BSON::OrderedHash代替：

```
doc = BSON::OrderedHash.new
doc['sym'] = 'GOOG'
doc['date'] = 20101005
@ticks.find(doc)
```

一定要检查使用的语言是否支持有序词库，如果不支持，则这种语言的MongoDB驱动程序会提供有序的备选。

数组

数组给文档模型更多的支持。如我们在电子商务事例中看到的，数组用于存储字符串列表、对象IDs，甚至其他的文档。数组提供了丰富又易于理解的文档。按理说，MongoDB易于查询和索引数组类型，而事实上，最简单的数组查询看起来就像查询任何其他文档类型，如表5.4所示。

表 5.4 数组操作符

操作符	描述
\$elemMatch	如果提供的所有词语在相同的子文档中，则匹配
\$size	如果子文档数组大小与提供的文本值相同，则匹配

让我们看看数组的实际操作。再次使用商品标签。这些标签代表一个简单的字符串列表：

```
{
  _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",
  tags: ["tools", "equipment", "soil"]
}
```

用标签"soil"查询商品是很简单的，和查询单一文档值的语法相同：

```
db.products.find({tags: "soil"})
```

重要的是，这种查询可以利用索引上的标签字段。如果建立所需索引并使用explain()运行查询，可以看到B-tree光标被使用：

```
db.products.ensureIndex({tags: 1})
db.products.find({tags: "soil"}).explain()
```

当我们需要对数组查询有更多的控制时，我们可以在特定位置的数组使用点符号查询值。以下所示的是如何限制上一个查询的第一个商品标签：

```
db.products.find({'tags.0': 'soil'})
```

这种查询方式不会有多大的意义。如果我们处理用户地址，就可能需要表示这些数字的子文档：

```
{
  _id: ObjectId("4c4b1476238d3b4dd5000001")
  username: "kbanker",
  addresses: [
    {
      name: "home",
      street: "588 5th Street",
      city: "Brooklyn",
      state: "NY",
      zip: 11215
    },
    {
      name: "work",
      street: "1 E. 23rd Street",
      city: "New York",
      state: "NY",
      zip: 10010
    }
  ]
}
```

我们可能规定数组的第0个元素始终是用户的配送地址。然而，要查找配送地址始终在纽约的所有用户时，我们可以再次指定第0个位置结合点目标state字段：

```
db.users.find({'addresses.0.state': 'NY'})
```

很容易理解，写产品代码时不应该指定元素。很容易忽略此地址并单独指定一个字段。如果列表中的任意地址属于New York，那么下面的查询会返回用户文档：

```
db.users.find({'addresses.state': 'NY'})
```

和以前一样，我们会想要虚线字段的索引：

```
db.users.ensureIndex({'addresses.state': 1})
```

注意，无论字段指向子文档还是子文档的数组，我们都使用相同的点符号。点符号很强大，并且其一致性令人放心，但用于单个或多个自对象数组属性查询时会出现歧义。例如，假如想获取家庭住址在纽约的用户列表，你能否想出以下查询方式？

```
db.users.find({'addresses.name': 'home', 'addresses.state': 'NY'})
```

这个查询的问题是字段引用并不局限于一个单一的地址。换句话说，这种查询匹配的地址指定一个是家、一个是纽约，但是我们想要的两个属性适用于同一个地址。幸运的是，有个查询运算符可以做到这点。同一个子文档限制多个条件时，我们使用\$elemMatch运算符，可以合理满足这种查询要求：

```

db.users.find({
  'addresses': {
    '$elemMatch': {
      'name': 'home',
      'state': 'NY'
    }
  }
})

```

从逻辑上讲，仅当我们需要匹配多个或者更多属性的子文档才使用\$elemMatch。

通过大小查询数组

仅仅是数组运算符才讨论\$size运算符。这种运算符允许通过自身大小查询数组。例如，如果想查找拥有三个地址的所有用户，就可以使用\$size运算符：

```

db.users.find({'addresses': {$size: 3}})

```

像这样写，\$size运算符不使用索引和限制准确匹配（我们不能指定size的范围）。因此，如果我们需要基于数组的大小执行查询，我们可以在文档自身的属性里缓存size并手动更新它。比如，我们可以考虑添加address_length字段到用户文档，然后在这个字段上建立一个索引，指定范围并精确匹配查询。一种可能的解决方案是使用聚合框架，我们将在第6章学习。

JavaScript 查询运算符

到目前为止，如果还是不能表示查询描述的工具，就可能需要编写一些JavaScript代码。可以使用特定的\$where运算符传递JavaScript表达式应对任何查询。

- \$where 执行任意JavaScript来选择文档。

在JavaScript范围内，关键字this指向当前文档。让我们操作一个实例：

```

db.reviews.find({
  '$where': "function() { return this.helpful_votes > 3; }"
})

```

还有一种简单表达式的缩写形式：

```

db.reviews.find({'$where': "this.helpful_votes > 3"})

```

我们可能不想使用这种查询工作，我们可以简便地使用其他查询运算符表达它。JavaScript表达式不能使用索引，并且带来大量费用，因为它们必须在JavaScript解释器的上下文中评估并且是单线程的。基于这些原因，当使用其他查询运算符无法不能表示查询时，就应该使用JavaScript查询。使用JavaScript表达式时至少结合一个其他查询运算符。使用其他查询运算符会消减结果集，减少必须加载到JavaScript上下文的文档数目。让我们看一个简单的例子，了解这样做的意义。

假设我们计算过每个用户的评价可靠性因素。本质上这还是一个整数，当用户的评分增加时，结果是更标准化的评价。假如想查询特定用户的评论并且返回标准评分高于3的，则以下代码可展示如何查询：

```
db.reviews.find({
  'user_id': ObjectId("4c4b1476238d3b4dd5000001"),
  '$where': "(this.rating * .92) > 3"
})
```

这个查询要符合这两条：一是它使用标准查询索引user_id字段，二是利用一个绝对超过其他查询运算符功能的JavaScript表达式。记住，有时候使用聚合框架会更简单。

除了意识到随之而来的性能损失，还必须注意JavaScript注入攻击的可能性。每当允许用户直接输入代码到JavaScript查询，就可能产生注入攻击。例如，当用户直接在排序查询中提交了一个用户表单和值时。如果用户设置属性值或者数值，则这种查询是不安全的：

```
@users.find({'$where' => "this.#{attribute} == #{value}"})
```

这种情况下，属性值和数值被插入成字符串，然后认定为JavaScript。这种方式是危险的，因为用户发送的值中可能会包含JavaScript代码，让他们获取其他数据集合。这将导致严重的安全漏洞，恶意用户可能会看到其他用户的数据。一般情况下，我们总是要假设用户可能会发送恶意数据和相应计划。

正则表达式

在本章开头，我们就接触并使用过查询正则表达式。在那个例子中，我们使用了前缀表达式/^Ba/，查找以Ba开头的姓，而且，这种查询使用了索引。事实上，还有更多的可能性。MongoDB使用Perl兼容正则表达式编译(PCRE; <http://mng.bz/hxmh>)，对正则表达式提供了全面的支持。

以下是\$regex运算符的概述。

■ \$regex 匹配元素对应提供给regex（正则表达式）项。

除了刚刚描述的前缀样式查询，正则表达式查询不能使用索引并且执行的时间比大多数选择器更长。我们建议节制性地使用它们。以下代码用于查询出包含最好或者最差词语的用户评论文本。注意我们要使用i正则表达式标签以指示区分大小写。

```
db.reviews.find({
  'user_id': ObjectId("4c4b1476238d3b4dd5000001"),
  'text': /best|worst/i
})
```

使用区分大小写标签会带来不利后果：它拒绝使用任何索引。在MongoDB中总是区分大小写。如果我们想在大量文档中使用区分大小写搜索，就需要使用提供新建文本搜索功能的2.4版本或者更高版本，或者集成外部的文本搜索引擎。请参阅第9章的MongoDB搜索功能的说明。

如果使用的语言含有自带正则表达式类型，就可以使用自带正则表达式对象执行查询。我们可以像这样在Ruby中表示相同的查询：

```
@reviews.find({
  :user_id => BSON::ObjectId("4c4b1476238d3b4dd5000001"),
  :text => /best|worst/i
})
```

尽管本地定义了正则表达式，但还是要要在MongoDB服务上做出评估。如果从一个本地不支持的正则表达式类型环境中查询，就可以使用特殊的\$regex和\$options运算符。通过命令使用这些运算符，我们就可以通过另一种方式表示查询：

```
db.reviews.find({
  'user_id': ObjectId("4c4b1476238d3b4dd5000001"),
  'text': {
    '$regex': "best|worst",
    '$options': "i"
  }
})
```

MongoDB是一个区分大小写的系统，当我们使用正则表达式时，除非使用/i修饰符（即/best|worst/i），否则搜索将与被搜索的字段大小完全匹配。有一点要注意，如果使用了/i，那么它会禁止使用索引。如果想做到索引内容不区分大小写搜索文档中的字符串、字段，那么就存储重复字段的内容专为强制小写搜索，或者使用MongoDB文本搜索功能，它可以与其他查询结合且提供了索引不区分大小写的搜索。

其他查询运算符

有些运算符不容易归类，但它们拥有自己的功能。第一个是\$mod，可以允许查询文档匹配modulo运算符；第二个是\$type，匹配BSON类型的值。在表5.5中包含两者的详情。

表 5.5 其他运算符总结

运算符	描述
\$mod[(quotient), (result)]	如果元素除以除数符合结果则匹配
\$type	如果元素的类型符合指定的 BSON 类型则匹配
\$text	允许在建立文本索引的字段上执行文本搜索

举个例子：使用下面的查询语句，\$mod允许查找所有被3整除的订单汇总：

```
db.orders.find({subtotal: {$mod: [3, 0]}})
```

我们可以看到\$mod运算符拥有一个数组含有两个值。第一个是除数，第二个预期的余数。查询技术上读做：“找到当总数除以3返回的余数为0的所有文档。”这是个人为的例子，但是论证了这个观点。如果最终使用了\$mod运算符，就不要使用索引。

第二个运算符\$type，按其BSON类型值匹配。不建议在同一个集合使用相同的字段存储多个类型，如果出现这种情况，那么有个查询运算符可用来根据类型进行测试。

表5.6所示为MongoDB中用于与每个元素类型相关联的数字类型。其中例子展示出类型的成员如何出现在JavaScript控制台。例如，其他MongoDB驱动程序可能用不同的方式存储等同的ISODate对象。

表 5.6 BSON 类型

BSON 类型	stype 数字	示例
Double	1	123.456
String (UTF-8)	2	"Now is the time"
Object	3	{ name:"Tim",age:"myob" }
Array	4	[123,2345,"string"]
Binary	5	BinData(2,"DgAAAEltIHNvbWUgYmluYXJ5")
ObjectId	7	ObjectId("4e1bdda65025ea6601560b50")
Boolean	8	true
Date	9	ISODate("2011-02-24T21:26:00Z")
Null	10	null
Regex	11	/test/i
JavaScript	13	function() {return false;}
Symbol	14	Not used; deprecated in the standard
Scoped JavaScript	15	function () {return false;}
32-bit integer	16	10
Timestamp	17	{ "t" : 1371429067, "i" : 0 }
64-bit integer	18	NumberLong(10)
Maxkey	127	{ "\$maxKey": 1 }
Minkey	255	{ "\$minKey" : 1 }
Maxkey	128	{ "maxkey" : { "\$maxKey" : 1 } }

在表5.6中有一对元素值得一提。maxkey 和 minkey用于在索引中插入相同的最大或者最小值的虚拟值。这意味着它可以强制文档排序为索引的第一个或最后一个项目。添加一个字段“aardvark”到集合中强制文档排序在前面，这种时光一去不复返了。大多数语言的驱动程序具有添加minkey或者maxkey类型的方法。

JavaScript的作用域和JavaScript在表中看起来相同，这仅仅是因为控制台不能展示作用域，这是键值对提供的JavaScript代码片段的字典。作用域指根据上下文执行函数。另一方面，从该函数可以看到变量在作用域字典中的定义和使用它们的执行过程。

最后，符号类型没有代表性。因为在大多数语言中都不会使用它，或者仅当语言有独特的类型“keys”时才会使用它。比如，在Ruby中，foo 和 :foo是不同的，:foo是一种符号。

Rubym驱动程序存储任何关键字为符号类型。

BSON符号类型

就查询而言，MongoDB服务对待BSON符号类型和对待字符串一样，只有当文档检索到一个独特的符号类型时，才会映射到相应的语言类型。注意，符号类型在最新的BSON规范中(<http://bsonspec.org>)被弃用，可能随时被丢弃。不管用什么语言写的数据库，可以在任何其他语言使用BSON检索它。

5.2.2 查询选择

所有的查询都需要查询选择器。即使查询为空，查询选择器实际上还是定义了一个空查询。当发出查询时，就有允许进一步限制结果集合的各种查询选项供选择。下面让我们看看这些选项。

映射

我们可以使用映射来选择子集的字，用来返回每个文档的查询结果集合。尤其在有大文档的情况下，使用映射可以最小化网络延迟和反序列化。对\$slice这个唯一运算符，下面进行了总结。

- \$slice 选择返回文档的子集。

映射通常定义为返回字段合集：

```
db.users.find({}, {'username': 1})
```

这种查询返回只包含字段username和_id的用户文档，排除其他所有字段这是特殊情况，默认情况下一直包含这2个字段。

在一些情况下，我们可能想指定排除相反的字段。这本书的用户文档包含了送货地址和付款方式，但我们通常不需要这些。要排除它们，就添加这些字段到映射值：

```
db.users.find({}, {'addresses': 0, 'payment_methods': 0})
```

在映射中，尽管_id字段是一种特殊情况，但也是要么包含要么排除。我们可以用同样的方式排除_id字段，通过设置0值到映射文档。

注意，除了包含和排除字段，我们可以将返回值的范围存储到一个数组。例如，我们可能要在商品自身的文档内存储商品的评论。这种情况下，我们坚持对评论分页，这时就可以使用\$slice运算符。返回前12条评论或者返回最后5条评论，就可这样使用\$slice：

```
db.products.find({}, {'reviews': {$slice: 12}})
db.products.find({}, {'reviews': {$slice: -5}})
```

`$slice`可以取两个元素数组，其中的值分别表示返回跳过和限制的页数。下面是如何跳过前24条评论并且限制评论为12条的代码：

```
db.products.find({}, {'reviews': {$slice: [24, 12]}})
```

最后注意，使用`$slice`后不会阻止其他返回字段。如果要在文档中限制其他字段，就必须明确这样做。例如，这是修改以前的查询，仅返回评论和评分：

```
db.products.find({}, {'reviews': {'$slice': [24, 12]}, 'reviews.rating': 1})
```

排序

当我们刚接触这一章时，我们只对任何查询结果的一个或多个字段按照升序或者降序排序。简单地通过评分对评论排序，如由最高分到最低分降序排列，像这样：

```
db.reviews.find({}).sort({'rating': -1})
```

当然，也可以按照有益的评论和评分排序：

```
db.reviews.find({}).sort({'helpful_votes': -1, 'rating': -1})
```

对这样的复合排序，顺序无关紧要。请注意，JSON通过命令排序。因为Ruby哈希是无序的，所以我们在Ruby数组中注明数组排序顺序，例如：

```
@reviews.find({}).sort(['helpful_votes', -1], ['rating', -1])
```

在MongoDB中指定排序是简单明了的，懂得如何建立索引是提升排序效率的关键。我们将在第8章继续学习，如果现在感觉使用排序很困难，则可以跳到前面学习。

跳过和限制

`skip`和`limit`已经没有什么神秘的了，这些查询选项总是如期望运行。但我们应小心使用大数值`skip`（值大于10 000），因为这种查询服务要求扫描的文档数等于`skip`值。假如我们正在对百万个文档按日期降序进行分页排序，每页有10个结果。这意味着查询显示第50 000页面将包含50万个`skip`值，这种效率非常低。更好的策略是完全忽略`skip`并且将范围添加到查询，指示下一个结果设置的位置。因此，这样的查询

```
db.docs.find({}).skip(500000).limit(10).sort({'date': -1})
```

变成这样：

```
previous_page_date = new Date(2013, 05, 05)
db.docs.find({'date': {'$gt': previous_page_date}}).limit(10).sort({'date': -1})
```

第二种查询浏览项目比第一种少。这里有一个潜在的问题是，如果`date`对每个文档不是唯一的，那么相同的文档可能会显示多次。有多种处理解决方案，把解决方案和练习留给读者。

这儿还有另外一套可以执行MongoDB数据的查询类型：地理空间查询，用于索引和检索地理位

置和几何数据，经常用于映射和位置感知应用程序。

5.3 总结

Summary

查询弥补了MongoDB接口的关键一角。当我们已经粗略阅读过本章的内容后，鼓励大家把查询机制放到测试中实战。如果不能确定查询运算符的特定组合如何为我们提供服务，那么shell就是最好的测试工具。

MongoDB还支持元运算符的查询修饰符，能让我们修改输出或者查询的性能。我们可以在这里找到更多说明资料：<http://docs.mongodb.org/manual/reference/operator/query-modifier/>。

从现在开始，我们会一直使用MongoDB查询，在接下来的两章得到更好的练习。我们会使用到聚合、文档更新和删除操作。查询在绝大多数更新中扮演关键角色，我们可以期待对于查询语言有更多探索。

本章内容

- 电商数据模型上的聚合
- 聚合框架细节
- 性能和限制
- 其他聚合功能

在前面一章我们看了如何使用MongoDB的类JSON查询语言来执行查询操作，比如根据ID、名字或者排序等。本章将会扩展这个主题，包括更多复杂的查询、使用MongoDB的聚合框架。聚合框架是MongoDB的高级查询语言，它允许我们通过转换和合并由多个文档中的数据来生成新的在单个文档里不存在的文档信息。例如，我们可以使用聚合框架根据月份来确定销售、根据产品来确定销售，或者根据用户来确定订单总数。这些对于关系型数据库来说都非常简单，我们可以把MongoDB的聚合框架等价于SQL的GROUP BY语句。虽然我们也可以通过MongoDB的map reduce功能或者使用程序代码计算出这些信息，但是聚合框架允许我们定义一系列文档操作，然后在单个调用里作为数组发送给MongoDB，让我们更容易地完成这些工作。

本章里，我们会展示使用电商数据模型的本书其余部分也会使用的许多例子，然后提供详细的聚合框架操作过程，以及不同的操作选项。在本章结束的时候，我们会有几个聚合框架关键知识点以及如何在电商数据模型里使用它们的对应例子。我们不会介绍聚合电商数据模型操作的细节，因为这就是聚合框架要完成的工作：它提供了比我们期望更多的灵活查询数据的方法。

到目前为止，我们已经设计过数据模型以及支持快速查询的数据库，还有响应式网站性能。聚合框架也可以帮助处理电商网站需要的实时信息，而且还提供了许多想从已有的数据里知道的各种问题的答案，但是这可能需要处理大量的数据。

图6.1所示为电商数据模型，每个人的购物车包含一个或多个商品，包括商品名称、价格、数量等。图6.2所示为聚合框架操作，聚合框架操作是一个管道，由一系列操作组成，每个操作都对前一个操作的结果进行转换。图6.3所示为聚合框架操作的结果，聚合框架操作的结果是一个文档数组，每个文档代表一个聚合操作的结果。

MongoDB 2.6、MongoDB 3.0 中的聚合

MongoDB聚合框架，第一次在MongoDB 2.2引入，每次新版本发布都会更新。本章包含了MongoDB 2.6的功能，第一次在2014年4月可用；MongoDB 3.0与MongoDB 2.6使用了相同的聚合框架，2.6版本加入了许多更新，以及改善聚合框架功能的操作符。如果你在使用更早期的MongoDB，就应该升级到2.6版本或者以后的版本来运行本章的例子代码。

6.1 聚合框架概览

Aggregation framework overview

为调用聚合框架就要定义一个管道（见图6.1）。聚合管道（aggregation pipeline）里的每一步输出都作为下一步的输入。每一步都在输入文档执行单个操作并生成输出文档。

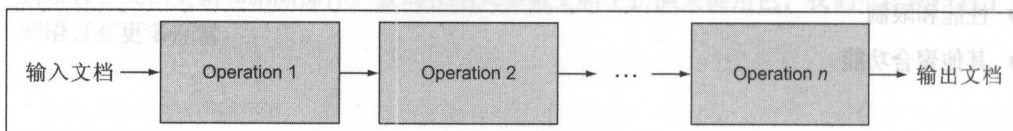


图6.1 聚合管道：一个操作的输出作为下一个操作的输入

聚合管道操作包含下面几个部分：

- `$project`——指定输出文档里的字段（项目化）。
- `$match`——选择要处理的文档，与`find()`类似。
- `$limit`——限制传递给下一步的文档数量。
- `$skip`——跳过一定数量的文档。
- `$unwind`——扩展数组，为每个数组入口生成一个输出文档。
- `$group`——根据key来分组文档。
- `$sort`——排序文档。
- `$geoNear`——选择某个地理位置附近的文档。
- `$out`——把管道的结果写入某个集合（2.6版新增）。
- `$redact`——控制特定数据的访问（2.6版新增）。

如果阅读过上一章关于构建MongoDB查询的内容，就会发现绝大部分操作符看起来都是相似的。因为绝大部分聚合框架操作符与MongoDB查询的函数类似。应该确保自己已经很好地理解了5.2节MongoDB查询语言的内容。

这个例子代码定义了一个聚合框架管道，包含一个匹配、组合排序：

```
db.products.aggregate([ {$match: ...}, {$group: ...}, {$sort: ...} ] )
```

这一系列操作如图6.2所示。

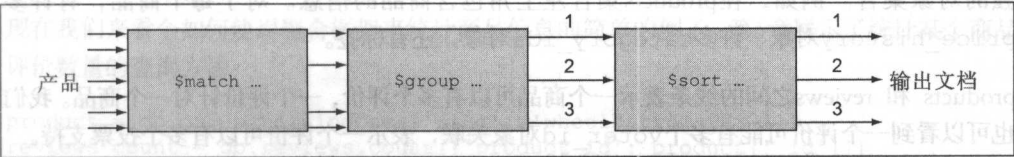


图 6.2 聚合框架管道的例子

如图所示，代码定义了管道，其中

- 整个商品集合传递给\$match操作，只从输入集合里选择一部分文档。
- \$match的输出结果传递给\$group操作符，然后根据key来进行排序，提供新的信息，比如求综合和求平均值。
- \$group的输出结果传递给\$sort操作符，然后将排序结果返回给最终的结果。

如果对于SQL的GROUP BY语句十分熟悉，就会知道它提供了总结信息。表6.1提供了SQL命令和聚合框架操作符对比的详细信息。

表 6.1 SQL 对比聚合框架

SQL 命令	聚合框架操作符
SELECT	\$project
	\$group functions: \$sum, \$min, \$avg, etc.
FROM	db.collectionName.aggregate(...)
JOIN	\$unwind
WHERE	\$match
GROUP	BY \$group
HAVING	\$match

下一节里，我们将会详细了解如何在电商数据模型里使用聚合框架。首先，我们要看一下如何使用聚合框架在Web页面提供商品的摘要信息。然后，我们再看看聚合框架如何应用到其他非Web程序里处理大量的数据，提供有价值的信息，比如找到曼哈顿上城区最高消费者。

6.2 电商聚合例子

E-commerce aggregation example

本节里我们会为电商数据库编写一些聚合查询的例子，回答大家关于使用聚合框架的许多疑问。在继续之前，我们先来复习一下电商网站e-commerce的数据模型。

图6.3所示为电商数据模型。每个大的箱子表示一个数据模型中的集合，包括products、reviews、

categories、orders、users。在每个集合里，我们展示了文档的结构，指定任意的数组作为单独的对象集合。例如，在products集合左上角包含商品的信息。对于每个商品，有许多price_history对象、许多category_ids对象，还有标签。

products 和 reviews之间的线条表示一个商品可以有多个评价，一个评价针对一个商品。我们也可以看到一个评价可能有多个voter_id对象关联，表示一个评价可以有多个投票支持。

随着数据模型的增长，这种模型会非常有帮助，因为很难记住多个集合之间的潜在关系，包括每个集合的内部细节。它在帮助确定回答何种问题时也非常有帮助。

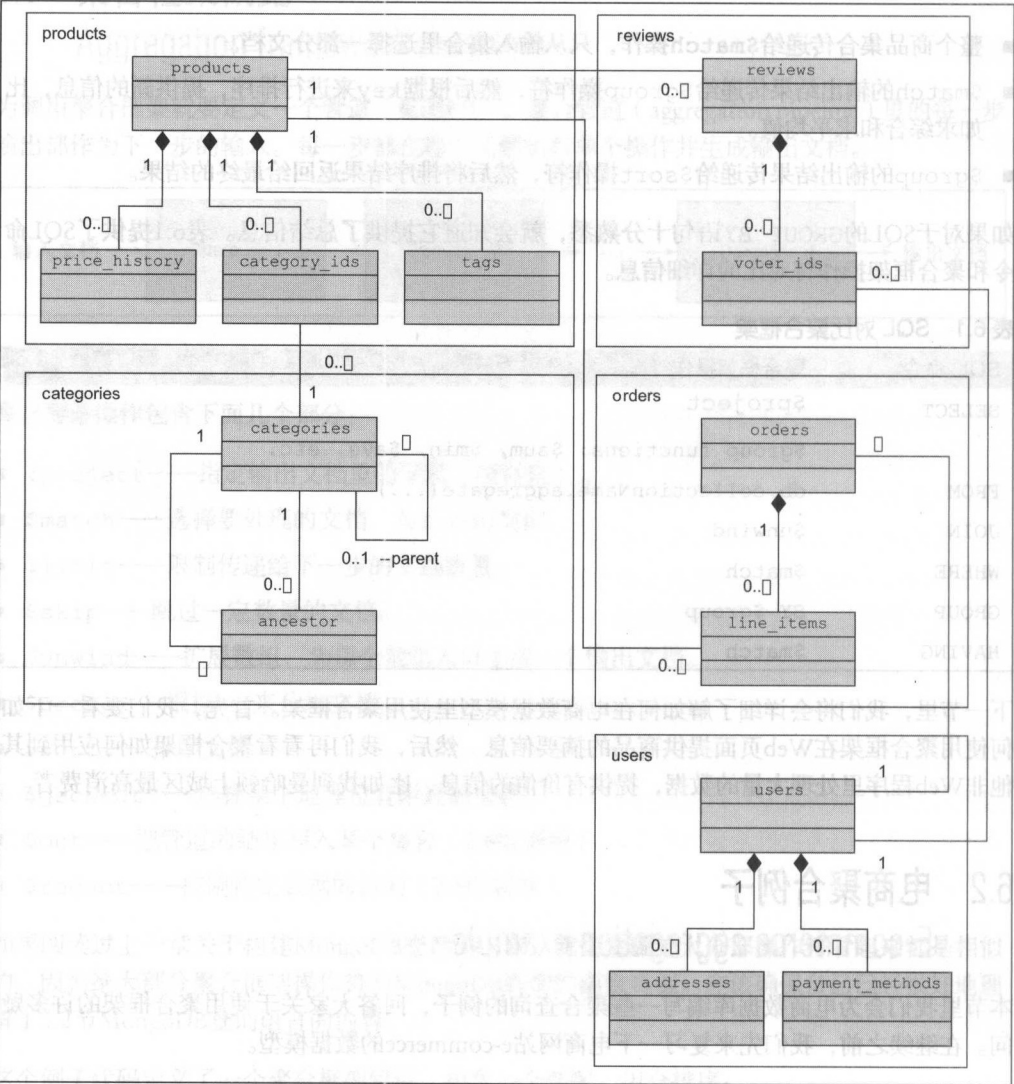


图 6.3 电商集合和关系的数据模型

6.2.1 商品、类别和评价

现在来看个如何使用聚合框架来统计商品信息的简单的例子。第5章展示了统计某个商品评价数量的查询方法：

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})
reviews_count = db.reviews.count({'product_id': product['_id']})
```

我们来看一下如何使用聚合框架。首先，我们来看一下如何统计所有商品的评价总数：

```
db.reviews.aggregate([
  { $group : { _id: '$product_id',          ← 根据 product_id 分组输入文档
               count: { $sum: 1 } } }      ← 计算每个商品的评价数量
]);
```

这个简单的操作符可以以文档结果的形式返回每个商品的评价信息：

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003982"), "count" : 2 } ← 为每个商品输出一个文档
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"), "count" : 3 }
```

这个例子中，我们有多文档作为\$group操作符的传入参数，但是对于每个_id就只有一个输出文档——此时每个唯一的product_id。\$group操作符会为每个输入商品文档添加一个1数字，为了每个product_id的计数统计。结果存放在输出文档的count字段里。

要注意的是，输入文档的字段通常会通过前缀美元符(\$)指定。在这个例子里，我们定义了_id的值是\$product_id，输入文档的product_id字段。

这个例子使用了\$sum函数来统计每个商品的评价文档数量，该数量增加1个，则每个product_id的商品评价数量count字段加1。\$group操作符支持许多功能，包括求平均值、最小值和最大值以及求和等。

更详细的功能介绍可以在6.3.2小节\$group操作符里看到。

接下来，为聚合管道添加一个操作，这样就可以选择唯一的商品来进行计算了：

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})

ratingSummary = db.reviews.aggregate([
  { $match : { product_id: product['_id'] } }, ← 只选择一个商品
  { $group : { _id: '$product_id',
               count: { $sum: 1 } } }
]).next(); ← 返回结果集的第一个文档
```

这个代码返回了我们感兴趣的文档，并把它赋值给ratingSummary变量。注意，聚合管道里的结果是个光标，指向允许处理任意数量结果、每次一个文档的指针。要查询单个文档，可以使用next()函数来返回光标中的第一个文档。

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"), "count" : 3 }
```

聚合光标: MongoDB 2.6 新增

在MongoDB 2.6之前的版本, 聚合管道返回的结果是最大值16MB的单个文档。从MongoDB 2.6以后, 我们可以使用光标处理任意大小的结果集。光标是默认的shell命令的返回值。但是为了避免破坏现有的程序, 默认的文档限制仍然是16MB。要在程序里使用光标, 我们可以重写默认设置来指定希望的光标。可以参看第6.5节的“聚合游标选项”的内容, 了解从聚合管道里返回的光标有什么其他的功能。

传递给\$match操作符的参数: {'product_id': product['_id']}应该看起来很熟悉。它们与第5章里使用计算商品评价的数量参数一样。

```
db.reviews.count({'product_id': product['_id']})
```

这些参数在前一章的5.1.1一节里详细介绍了。这里介绍的绝大部分操作符对于\$match都可用。

非常重要是\$match要在\$group前面。我们不能搞反了顺序, 把\$match 放到\$group后面, 返回的结果是一样的。但是这样做会让MongoDB计算所有商品的评价数量, 然后抛弃所有的, 只保留一个结果。通过\$match前置, 就可以大大减少\$group操作符要处理的文档数量。

既然我们已经获得商品的总评价数量, 那么现在就可以看看如何计算商品的平均评分了。这个主题超出了第5章查询语言的内容范畴。

计算评价的平均值

要计算商品评价的平均评分, 就需要使用和之前例子一样的管道, 添加一个新字段:

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})

ratingSummary = db.reviews.aggregate([
  {'$match': {'product_id': product['_id']}},
  {'$group': {'_id': '$product_id',
    average: {'$avg': '$rating'},
    count: {'$sum': 1}}}
]).next();
```

← 计算商品的平均评分

之前的例子返回单个文档, 然后复制给变量ratingSummary, 文档的内容如下:

```
{
  "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "average" : 4.333333333333333,
  "count" : 3
}
```

这个例子使用\$avg函数来计算商品的平均评分。

注意, 平均评分在\$avg函数里使用'\$rating'指定。这种做法这里同样也适用, 如下代码的字段为\$group-id值指定的字段:

```
_id:'$product_id'.
```

细分评分

接下来，我们来扩展一下商品的评分统计信息，展示分类评分信息。大家可能已经在某些购物网站看到过图6.4所示界面。我们可以看到，有5个评分是满分5分，有2个评分是4分，有1个评分是3分。

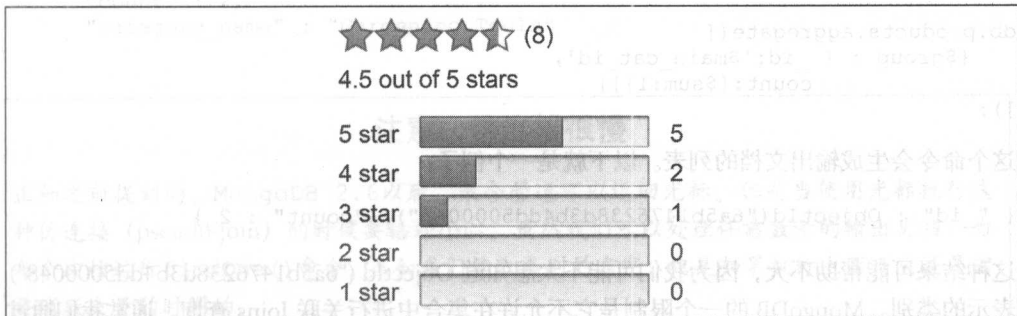


图 6.4 评价统计

使用聚合框架就像用一个命令一样来计算这些信息。这种情况下，首先使用\$match来选择要计算的单个商品，和之前的例子一样。接下来，我们进行评分分类并计算每个评分的数量。

下面是实现此功能的聚合命令代码：

```
countsByRating = db.reviews.aggregate([  
  {$match : {'product_id': product['_id']}},  ← 选择商品  
  {$group : { '_id': '$rating',  
    count: {$sum:1}}}  ← 根据评分值分组  
]).toArray();  ← 把结果光标转换为数组  
                ← 为每个评分统计评价数
```

如代码所示，我们使用\$sum函数计算出了数量；这次，我们为每个评分计算出了单独的数量。另外要注意，聚合调用的结果是个光标，我们已经把它转换为数据，并且赋值给countsByRating变量了。

SQL query 查询

对于熟悉SQL语句的开发者来说，等价的SQL查询代码如下所示：

```
SELECT RATING, COUNT(*) AS COUNT  
FROM REVIEWS  
WHERE PRODUCT_ID = '4c4b1476238d3b4dd5003981'  
GROUP BY RATING
```

聚合调用会生成如下类似的数组：


```
[ { "_id" : 5, "count" : 5 },
  { "_id" : 4, "count" : 2 },
  { "_id" : 3, "count" : 1 } ]
```

连接集合

接下来，假设我们要检查数据库的内容，计算每个类别下的商品数量。记住，商品只有一个主类别。聚合命令如下所示：

```
db.products.aggregate([
  {$group : { _id:'$main_cat_id',
              count:{$sum:1}}}
]);
```

这个命令会生成输出文档的列表。以下就是一个例子：

```
{ "_id" : ObjectId("6a5b1476238d3b4dd5000048"), "count" : 2 }
```

这种结果可能帮助不大，因为我们可能不太想知道 ObjectId ("6a5b1476238d3b4dd5000048") 表示的类别。MongoDB 的一个限制是它不允许在集合中进行关联 Joins 查询。通常我们通过去范式来解决这个问题——让它包含、通过分组或者冗余定义的电商网站期望的字段。例如，在订单集合里，每个文档包含一个商品名字，所以不需要其他调用来获取集合中的商品名字。

但要记住，聚合框架经常用来生成自定义查询的统计报告，这些统计报告也许无法提前获得。我们也许要知道需要多少数据违反范式才不至于复制太多的冗余数据。冗余数据可能会增加数据库的存储空间，使更新复杂化（因为可能需要更新多个文档中的相同信息字段）。

虽然从 MongoDB 2.6 开始 MongoDB 不允许自动化连接，但是有多种方式提供等价的 SQL 连接。一种选择是使用 forEach 函数来处理聚合命令返回的光标，并且使用伪连接（pseudo-join）添加名字。这是例子：

```
db.mainCategorySummary.remove({});
db.products.aggregate([
  {$group : { _id:'$main_cat_id',
              count:{$sum:1}}}
]).forEach(function(doc) {
  var category = db.categories.findOne({_id:doc._id});
  if (category !== null) {
    doc.category_name = category.name;
  }
  else {
    doc.category_name = 'not found';
  }
  db.mainCategorySummary.insert(doc);
});
```

从 mainCategorySummary 集合删除已有文档

读取类别

无法保证类别实际存在

向摘要集合插入组合结果

在这段代码里，假如存在文档，我们首先从 mainCategorySummary 删除所有的文档。要执行伪连接，就要处理每个结果文档，并且执行 findOne() 调用来读取类别名称。把类别名字添加到输出结果文档里后，再把结果插入到名为 mainCategorySummary 的集合里。我们将

会在下一章里讲解这个插入函数。

在mainCategorySummary集合上执行find()，返回的结果会包含每个类别信息。下面的findOne()命令展示了第一个结果的信息：

```
> db.mainCategorySummary.findOne();
{
  "_id" : ObjectId("6a5b1476238d3b4dd5000048"),
  "count" : 2,
  "category_name" : "Gardening Tools"
}
```

注意：伪连接很慢

正如之前提到的，MongoDB 2.6以后，聚合管道可以返回光标。但是当使用光标执行这种伪连接（pseudo-join）的时候要格外小心。虽然我们可以处理任意数量的输出文档，为每个文档运行findOne()命令，正如我们读取类别的名字，但是如果执行上百万次还是需要消耗大量的时间的。

\$OUT AND \$PROJECT

我们很快就会看到一个更快的连接操作符\$unwind，但是，我们先应该理解另外2个操作符：\$out 和 \$project。在前一个例子里，我们把聚合管道的结果集保存到一个名为mainCategorySummary集合里，然后使用代码处理每个文档。最后使用下面的代码保存文档：

```
db.mainCategorySummary.insert(doc);
```

使用\$out操作符，可以自动把聚合管道的输出结果保存到集合里。如果集合不存在，则\$out操作符将会创建一个集合，或者如果存在就会完全取代现有的集合。此外，如果创建新的集合失败的话，MongoDB不会修改之前的集合。例如，下面的代码将会把聚合管道的集合保存到一个命名的集合里：

```
mainCategorySummary:
```

```
db.products.aggregate([
  { $group : { _id : '$main_cat_id',
               count : { $sum : 1 } } },
  { $out : 'mainCategorySummary' } ])
```

把管道结果保存到 mainCategorySummary

\$project操作符允许我们过滤可以传递给管道下一个阶段的字段。虽然\$match允许我们通过限制文档数量传递下一步的数据量，但是\$project可以用来限制每个传给下一步文档的大小。限制每个文档的大小可以改善性能，尤其是在处理大文档并且只需要每个文档一部分数据的时候。下面是\$project操作符的例子，通过限制输出文档来调整每个产品文档的类别ID：

```
> db.products.aggregate([
... {$project : {category_ids:1}}
... ]);
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "category_ids" : [ ObjectId("6a5b1476238d3b4dd5000048"),
                    ObjectId("6a5b1476238d3b4dd5000049") ] }
{ "_id" : ObjectId("4c4b1476238d3b4dd5003982"),
  "category_ids" : [ ObjectId("6a5b1476238d3b4dd5000048"),
                    ObjectId("6a5b1476238d3b4dd5000049") ] }
```

现在我们来看看如何使用\$unwind操作符执行快速连接。

使用\$UNWIND 更快速连接

接下来我们将会看到聚合框架另外一个强大的功能特性：\$unwind操作。这个操作符允许我们扩展数组，为每个输入文档的数组生成一个输出文档。因此，它可以提供另外一种类型的MongoDB连接，这样我们就可以使用每个子文档来连接文档了。

早期，当商品只有一个主类别的时候，要计算每个类别的商品数量。现在假设我们要计算每个类别的商品数量，无论它是不是主类别。回忆一下图6.4展示的数据模型，每个商品都包含一个category_ids数组。\$unwind操作符允许我们使用每个数组元素来连接每个商品，为每个商品和category_id生成一个文档。我们可以通过category_id来进行数据统计。聚合命令列表6.1所示。

列表6.1 \$unwind使用类别id数组来连接每个商品文档

```
db.products.aggregate([
  {$project : {category_ids:1}},
  {$unwind : '$category_ids'},
  {$group : { '_id':'$category_ids',
              count:{$sum:1}}},
  {$out : 'countsByCategory'}
]);
```

只传递类别 ID 给下一步。
默认传递_id 特性
为每个 category_ids 中的入口项目创建一个文档
\$out 把聚合结果写入到集合 countsByCategory

聚合管道里的第一个操作符\$project，限制传递给下一阶段的字段属性，它对于使用\$unwind操作符非常重要。使用\$unwind将会为数组里的每个元素生成一个输出文档，当想要限制这些输出结果的大小时，\$project操作符的作用就体现出来了。如果文档其余的部分很大，而且数组包含了大量的元素，管道里就会有大量的结果集传递给下一步。MongoDB 2.6之前的版本对此会导致出错，但是对于MongoDB 2.6以后的版本，大文档也会降低管道的处理速度。我们也可以使用磁盘来存储输出结果，但这样也会进一步让管道降速。

管道里的最后一个操作符是\$out，用于把结果保存到名为countsByCategory的集合里。以下是一个保存输出结果到countsByCategory集合的例子代码：

```
> db.countsByCategory.findOne()
{ "_id" : ObjectId("6a5b1476238d3b4dd5000049"), "count" : 2 }
```

一旦加载了新的集合countsByCategory，如果需要，就可以为集合里的每一行添加类别名称。下一章里，将会展示如何更新集合。

我们已经看到了如何使用聚合框架根据商品和类别来生成各种不同的统计结果。前一节也介绍了两个聚合管道里非常重要的操作符：\$group和\$unwind。我们已经了解\$out操作符了，它可以用来保存聚合查询的结果。现在，我们来看看关于用户和订单分析的有用的总结。我们也会介绍一些聚合功能，并展示这些例子。

6.2.2 用户和订单

编写本书第一版的时候，第一次在MongoDB 2.2引入了聚合框架，还没有正式发布。第一版有两个例子使用了map-reduce函数，通过用户分组评分，根据月份统计销售信息。例子根据用户分组统计了每个用户的评价数量，以及有帮助的投票。本书使用的聚合框架提供了更加简单的、直观的方法：

```
db.reviews.aggregate([
  { $group :
    { _id : '$user_id',
      count : { $sum : 1 },
      avg_helpful : { $avg : '$helpful_votes' } }
  ]
})
```

调用的返回结果如下所示：

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000003"),
  "count" : 1, "avg_helpful" : 10 }
{ "_id" : ObjectId("4c4b1476238d3b4dd5000002"),
  "count" : 2, "avg_helpful" : 4 }
{ "_id" : ObjectId("4c4b1476238d3b4dd5000001"),
  "count" : 2, "avg_helpful" : 5 }
```

根据年、月统计销售信息

下面的例子展示了2010年根据月和年来统计的订单数据。我们可以在6.6.2小节看到如何使用MongoDB map-reduce实现类似的功能，它需要18行代码来生成相同的统计数据。下面是使用聚合框架生成的结果：

```
db.orders.aggregate([
  { $match: { purchase_data: { $gte: new Date(2010, 0, 1) } } },
  { $group: {
    _id: { year : { $year : '$purchase_data' },
          month: { $month : '$purchase_data' } },
    count: { $sum: 1 },
    total: { $sum: '$sub_total' } },
    { $sort: { _id: -1 } }
  ]
});
```

运行这个命令，可以看到结果如下所示：

```
{ "_id": { "year": 2014, "month": 11 },
  "count": 1, "total": 4897 }
{ "_id": { "year": 2014, "month": 10 },
  "count": 2, "total": 11093 }
{ "_id": { "year": 2014, "month": 9 },
  "count": 1, "total": 4897 }
```

在这个例子里，我们使用\$match操作符只选择2010年1月1日之后的订单。注意，在JavaScript里，1月是从0开始的，所以我们对于日期的限制是Date(2010,0,1)以后。匹配函数\$gte看起来应该很熟悉，因为在5.1.2节中介绍过。

对于\$group操作符，我们使用年和月组合键来分组订单。虽然在集合里不经常使用组合键，但是在聚合框架里它们非常有用。在这个例子里，组合键由2个字段组成：year和month。我们也可以使用\$year和\$month函数从交易日期里抽取年、月字段。我们要计算订单的数量\$sum:1，还有计算订单的总和\$sum:\$sub_total。

管道里最后的操作是排序从最近到最老的月份数据结果。对于传递给\$sort的值也应该不陌生：与MongoDB查询sort()函数相同。注意组合键字段的顺序，_id会有问题。如果把月份放到年份前面，排序就会先月份然后年份。这看起来很奇怪，除非我们想要确定每月的趋势。

既然我们已经熟悉了聚合框架的基本操作，现在就来看一个更复杂的查询。

找到曼哈顿最好的客户

在5.1.2小节里，我们找到了曼哈顿所有的客户。现在我们来扩展查询，找出曼哈顿消费最高的用户。管道如图6.5所示。注意，\$match是管道里的第一步，用来减少管道处理的文档数量。

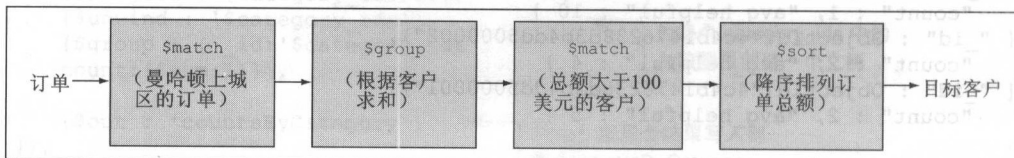


图 6.5 选择目标用户

查询包含以下步骤：

- \$match——查找快递到曼哈顿的订单。
- \$group——为每个客户求和。
- \$match——选择总额超过100美元的客户。
- \$sort——按降序排列结果。

我们将使用更简单的开发和测试管道方法。首先，定义每一步使用的参数：

```
upperManhattanOrders = {'shipping_address.zip': {$gte: 10019, $lt: 10040}};
sumByUserId = {_id: '$user_id',
```

```
total: {$sum: '$sub_total'}, });
orderTotalLarge = {total: {$gt: 10000}};
sortTotalDesc = {total: -1};
```

这些命令定义了要传递给聚合管道每一步的参数。这样做便于理解管道，因为嵌套的JSON对象难以理解。基于这些定义，整个管道的调用会如下所示：

```
db.orders.aggregate([
  {$match: upperManhattanOrders},
  {$group: sumByUserId},
  {$match: orderTotalLarge},
  {$sort: sortTotalDesc}
]);
```

我们可以单独或者组合测试管道中的每一步。例如，我们来运行管道，统计所有的客户：

```
db.orders.aggregate([
  {$group: sumByUserId},
  {$match: orderTotalLarge},
  {$limit: 10}
]);
```

下面的代码将会展示10个用户的列表：

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000002"), "total" : 19588 }
```

假设我们决定要保留订单的数量，那么要修改sumByuserId的值：

```
sumByUserId = { _id: '$user_id',
  total: {$sum: '$sub_total'},
  count: {$sum: 1}};
```

返回前面的聚合命令，我们会看到下面的结果：

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000002"),
  "total" : 19588, "count" : 4 }
```

这样构建聚合管道，可以允许我们简单地开发、迭代以及测试管道，而且也易于理解。一旦对结果满意，就可以添加\$out操作符以便把结果保存到新的集合中了，因此，可以通过不同的应用来非常简单地访问这个结果：

```
db.orders.aggregate([
  {$match: upperManhattanOrders},
  {$group: sumByUserId},
  {$match: orderTotalLarge},
  {$sort: sortTotalDesc},
  {$out: 'targetedCustomers'}
]);
```

我们已经学习了聚合框架，它可以帮助我们突破许多数据库设计的限制，允许我们扩展前一章里学习的知识来分析和聚合数据。我们还学习了聚合管道以及管道里核心的操作符，包括\$group和\$unwind。接下来，我们将会详细看一下每个聚合操作符，并且介绍如何使用它

们。正如我们之前提到的，如果读过前一章内容，在这里就会发现很多内容都很熟悉。

6.3 聚合管道操作符

Aggregation pipeline operators

聚合框架支持10个操作符：

- `$project`——指定要处理的字段。
- `$group`——根据指定的key进行分组。
- `$match`——选择要处理的文档，与`find(...)`类似。
- `$limit`——限制传递给下一步的文档数量。
- `$skip`——不传递给下一步的文档数量。
- `$unwind`——扩展数组，为每个数组元素生成一个输出文档。
- `$sort`——对文档排序。
- `$geoNear`——选择地理位置附近的文档。
- `$out`——把管道的结果输入一个集合里（2.6版本新增的）。
- `$redact`——控制特定数据的访问（2.6版本新增的）。

下面的小节详细介绍了这些操作符。对于绝大部分应用，`$geoNear`和`$redact`操作符用得很少，本章就不做介绍了。可以在<http://docs.mongodb.org/manual/reference/operator/aggregation/>查看更多详细内容。

6.3.1 `$project`

`$project` 操作符包含第5章介绍的所有查询映射功能，而且更多。下面的查询就是基于5.1.2节读取用户的姓和名的例子：

```
db.users.findOne(  
  {username: 'kbanker',  
   hashed_password: 'bd1cfa194c3a603e7186780824b04419'},  
  {first_name:1, last_name:1}  
)
```

返回姓名的
投影对象

我们可以使用如下代码实现与前面例子相同的查询条件和映射对象：

```
db.users.aggregate([  
  {$match: {username: 'kbanker',  
            hashed_password: 'bd1cfa194c3a603e7186780824b04419'}},  
  {$project: {first_name:1, last_name:1}}  
)
```

返回姓名的投
影管道操作符

除了与前面查询映射使用相同的特性，我们还可以使用大量文档的重塑功能。因为文档太多，而且可以用来定义\$group操作符的_id字段在6.4节里已有介绍，它主要关注重塑文档。

6.3.2 \$group

\$group操作符主要用于聚合管道。此操作符可以处理多个文档的聚合数据，提供诸如min、max、average的统计功能。对于熟悉SQL的读者来说，\$group操作符等价于SQL的GROUP BY语句。

\$group聚合功能的完整列表如表6.2所示。

我们可以告诉\$group操作符如何通过定义_id字段来分组文档。然后\$group操作符根据指定的_id字段分组输入文档，提供每组文档的聚合信息。下面的例子如6.2.2小节所示，我们可以通过月和年来统计销售信息：

```
> db.orders.aggregate([
...   {$match: {purchase_data: {$gte: new Date(2010, 0, 1)}}},
...   {$group: {
...     _id: {year : {$year : '$purchase_data'},
...           month: {$month : '$purchase_data'}},
...     count: {$sum:1},
...     total: {$sum:'$sub_total'}}},
...   {$sort: {_id:-1}}
... ]);
{ "_id" : { "year" : 2014, "month" : 11 },
  "count" : 1, "total" : 4897 }
{ "_id" : { "year" : 2014, "month" : 8 },
  "count" : 2, "total" : 11093 }
{ "_id" : { "year" : 2014, "month" : 4 },
  "count" : 1, "total" : 4897 }
```

当为分组定义_id字段时，我们可以使用一个或者更多存在的字段，或者可以使用6.4节里介绍的文档重塑函数。这个例子演示了使用2个重塑函数：\$year和\$month。

只有_id字段可以使用重塑功能（reshaping）。\$group输出文档里的其他字段限制使用表6.2所示的\$group函数。

表 6.2 \$group 函数

\$group 函数	
\$addToSet	为组里唯一的值创建一个数组
\$first	组里的第一个值。只有前缀\$sort才有意义
\$last	组里最后一个值，只有前缀\$sort才有意义
\$max	组里某个字段的最大值
\$min	组里某个字段的最小值
\$avg	某个字段的平均值

\$group 函数

\$push	返回组内所有值的数组。不去除重复值
\$sum	求组内所有值的和

这些函数大多都浅显易懂，但这两个不太明显：\$push和\$saddToSet。下面的例子创建了客户端的列表，每个客户端包含一个商品数组，商品数组使用\$push函数创建：

```
db.orders.aggregate([
  {$project: {user_id:1, line_items:1}},
  {$unwind: '$line_items'},
  {$group: {
    _id: {user_id:'$user_id'},
    purchasedItems: {$push: '$line_items'}}}
]).toArray();
```

\$push 函数把对象添加到 purchasedItems 数组

前面的例子代码输出的结果如下：

```
{
  "_id" : {
    "user_id" : ObjectId("4c4b1476238d3b4dd5000002")
  },
  "purchasedItems" : [
    {
      "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
      "sku" : "9092",
      "name" : "Extra Large Wheel Barrow",
      "quantity" : 1,
      "pricing" : {
        "retail" : 5897,
        "sale" : 4897
      }
    },
    {
      "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
      "sku" : "9092",
      "name" : "Extra Large Wheel Barrow",
      "quantity" : 1,
      "pricing" : {
        "retail" : 5897,
        "sale" : 4897
      }
    }
  ],
  ...
}
```

\$addToSet与\$push

看到分组函数，你可能好奇\$saddToSet和\$push的区别。集合中的元素必须确保唯一。某个给定的值不能在集合里出现两次，而且可以通过\$saddToSet强制实行。而\$push操作符没有这个限制，集合中的值可以不唯一。因此，相同的元素可以在\$push创建的数组里多次出现。

我们再来继续学习一些看起来很熟悉的操作符。

6.3.3 \$match、\$sort、\$skip、\$limit

这4个操作符放在一起介绍，因为它们与第5章里介绍的查询函数相同。使用这些操作符，我们可以选择特定的文档、排序文档、跳过特定数量的文档、限制处理文档的数量。

把它们与第5章里介绍的查询语言对比，会发现它们的参数也一样。以下是5.1.1节里分页查询的例子代码：

```
page_number = 1
product = db.products.findOne({'slug': 'wheelbarrow-9092'})
reviews = db.reviews.find({'product_id': product['_id']}).
    skip((page_number - 1) * 12).
    limit(12).
    sort({'helpful_votes': -1})
```

聚合框架等价的查询代码如下所示：

```
reviews2 = db.reviews.aggregate([
    {$match: {'product_id': product['_id']}},
    {$skip: (page_number - 1) * 12},
    {$limit: 12},
    {$sort: {'helpful_votes': -1}}
]).toArray();
```

正如你看到的，两个版本的代码功能和输入参数是一样的。

一个例外之处是`find()` `$where`函数，它允许我们使用JavaScript表达式来选择文档。`$where`不能使用聚合框架的`$match`操作符。

6.3.4 \$unwind

我们在6.2.1小节里讨论快速连接的时候已经看到过`$unwind`。这个操作符通过为数组里的每个元素生成一个输出文档来扩展数组。主文档里的字段、每个数组元素的字段都被放入输出文档里。这个例子显示了`$unwind`之前和之后商品的类别。

```
> db.products.findOne({}, {category_ids:1})
{
  "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "category_ids" : [
    ObjectId("6a5b1476238d3b4dd5000048"),
    ObjectId("6a5b1476238d3b4dd5000049")
  ]
}
```

```
> db.products.aggregate([
...    {$project : {category_ids:1}},
```

```
...    {$unwind : '$category_ids'},
...    {$limit : 2}
...  ]);
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "category_ids" : ObjectId("6a5b1476238d3b4dd5000048") }
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "category_ids" : ObjectId("6a5b1476238d3b4dd5000049") }
```

现在我们来查看一个MongoDB v2.6中的新操作符\$out。

6.3.5 \$out

在6.2.2中，我们创建了一个管道来查询曼哈顿最好的客户。我们这里会再次使用这个例子，但是这次的管道输出文档使用\$out操作符保存在targetedCustomers集合里。\$out操作符必须是管道里的最后一个操作符：

```
db.orders.aggregate([
  {$match: upperManhattanOrders},
  {$group: sumByUserId},
  {$match: orderTotalLarge},
  {$sort: sortTotalDesc},
  {$out: 'targetedCustomers'}
]);
```

加载的结果数据必须符合集合的约束。例如，所有集合文档必须有唯一的_id。如果因为某些原因，管道失败了，无论是在\$out操作之前还是过程中，现有的集合都不能改变。在使用这个方法生成一个等价的SQL物化视图时要记住这一点^[1]。

MongoDB 的物化视图

绝大部分关系型数据库都提供了物化视图功能。物化视图提供了一种高效和易用的方式来访问提前生成的数据结果。通过提前生成这些信息，我们可以节约大量的生成需要数据的时间，而且也便于应用程序预处理信息。\$out操作的失败安全性是生成等价物理视图的关键。无论什么原因，如果生成新集合失败，都会保留之前的集合。如果我们期望其他也应该可以独立使用这个信息，则这是个非常重要的特性，就算集合过时也比丢失数据好多了。

我们现在已经看了主要的管道操作符。现在我们回到之前提到的主题上：重塑文档。

6.4 重塑文档

Reshaping documents

MongoDB的聚合管道包含许多可以用来重塑文档的函数，因此可以生成一个包含最初文档没有的字段的新文档。我们通常会与\$project操作符一起使用这些函数，但是也可以在为

^[1]【译者注】SQL Materialized views，物化视图，就是包含查询数据的视图。

\$group操作符定义_id时使用。

最简单的重塑功能就是把一个字段进行重命名，生成一个新字段。我们也可以通过修改或者创建一个新文档来重塑一个文档。例如，回到之前读取用户姓和名字段的例子，如果要创建一个同时包含两个字段first和last名为name的字段，可以使用如下代码实现：

```
db.users.aggregate([
  {$match: {username: 'kbanker'}},
  {$project: {name: {$first: '$first_name',
                    last: '$last_name'}}}
])
```

运行代码的输出结果如下所示：

```
{ "_id": ObjectId("4c4b1476238d3b4dd5000001"),
  "name": { "first": "Kyle",
            "last": "Banker" } }
```

除了重命名或者重新构建文档字段外，我们也可以使用不同的重塑函数来创建新的字段。重塑函数根据处理数据类型不同进行分组：字符串、算数运算、日期、逻辑、集合、其他类型。

接下来，我们将会详细看一下每组不同的函数，从执行字符串操作开始。

聚合框架重塑函数

有许多函数——似乎每次发布都会增加函数——允许我们在输入文档中执行许多不同的操作以生成新字段。

本节里我们会看一下各种不同的操作符，同时进行复杂重塑功能的实现。最新的可用函数列表可以参考MongoDB 官方文档 <http://docs.mongodb.org/manual/reference/operator/aggregation/group/>。

6.4.1 字符串函数

字符串函数如表6.3所示，允许我们操作字符串。

表 6.3 字符串函数

Strings	
\$concat	连接 2 个或者更多字符串为一个字符串
\$strcasecmp	大小写敏感的比较，返回数字
\$substr	获取字符串的子串
\$toLowerCase	转换为小写字符串
\$toUpperCase	转换为大写字符串

这个例子使用了三个函数——\$concat、\$substr、\$toUpper:

```
db.users.aggregate([
  { $match: { username: 'kbanker' } },
  { $project:
    { name: { $concat: ['$first_name', ' ', '$last_name!'] },
      firstInitial: { $substr: ['$first_name', 0, 1] },
      usernameUpperCase: { $toUpper: '$username' } }
  }
])
```

姓名使用
空格连接

名字的第一个
字符初始化

修改 username
为大写字母

运行代码的结果如下所示:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5000001"),
  "name" : "Kyle Banker",
  "firstInitial" : "K",
  "usernameUpperCase" : "KBANKER"
}
```

接下来我们来看下算术运算函数。

6.4.2 算术运算函数

算术运算函数包含标准的算术运算操作，如表6.4所示。

表 6.4 算术运算函数

算术运算	
\$add	求和
\$divide	除法
\$mod	求余数
\$multiply	乘积
\$subtract	减法

通常，算术运算函数允许我们对数字执行基本的计算，例如，加、减、乘、除。

接下来我们来看一些与日期相关的函数。

6.4.3 日期函数

日期函数如表6.5所示，使用现有的日期或者通过计算年、月、日等来创建新字段。

表 6.5 日期函数

日期	
\$dayOfYear	一年 365 天中的某一天
\$dayOfMonth	一月中的某一天
\$dayOfWeek	一周中的某一天，1 表示周日
\$year	日期的年份
\$month	日期的月份，1~12
\$week	一年中的某一周，0~53
\$hour	日期中的小时，0~23
\$minute	日期中的分钟，0~59
\$second	日期中的秒，0~59
\$millisecond	日期中的毫秒，0~999

我们已经在6.2.2小节里看过\$year和\$month的例子，使用它们来根据年份和月份统计销售数据。剩下的日期函数比较简单了，所以我们接下来就详细看一下逻辑函数。

6.4.4 逻辑函数

逻辑函数如表6.6所示，看起来很熟悉。绝大部分与第5章5.2节里总结的查询操作符类似。

表 6.6 逻辑函数

逻辑	
\$and true	与操作，如果数组里所有值都为 true，则返回 true
\$cmp	如果两个数相等就返回 0
\$cond if... then... else	条件逻辑
\$eq	两个值是否相等
\$gt	值 1 是否大于值 2
\$gte	值 1 是否大于等于值 2
\$ifNull	把 null 值/表达式转换为特定的值
\$lt	值 1 是否小于值 2
\$lte	值 1 是否小于等值 2
\$ne	值 1 是否不等于值 2
\$not.	取反操作
\$or	或，如果数组中有一个 true，就返回 true

\$cond函数与我们看到的其他函数不同，它允许复杂的操作：如果、然后、其他。这与很多语言里的三元操作符很像。例如x ? y : z，表示如果条件x为true就选择y，否则就选择z。

接下来是集合操作符，它允许我们使用不同的方式来比较集合的值。

6.4.5 集合操作符

集合操作符总结在表6.7中，允许我们比较两个数组的内容。使用集合操作符，可以比较两个数组、判断它们是否完全一样、是否有公共元素、是否有差别元素。如果需要使用这些函数，最简单的方式就是查看MongoDB的官方文档：<http://docs.mongodb.org/manual/reference/operator/aggregation-set/>。

表 6.7 集合函数

	集合
\$setEquals	如果两个集合的元素完全相同，则为 true
\$setIntersection	返回两个集合的公共元素
\$setDifference	返回第一个集合中与第二个集合不同的元素
\$setUnion	合并集合
\$setIsSubset	如果第二个集合为第一个集合的子集，则为 true
\$anyElementTrue	如果某个集合元素为 true，则为 true
\$allElementsTrue true	如果所有集合元素都为 true，则为 true

这是个使用\$setUnion函数的例子。假设我们已经有如下商品：

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "productName" : "Extra Large Wheel Barrow",
  "tags" : [ "tools", "gardening", "soil" ] }
{ "_id" : ObjectId("4c4b1476238d3b4dd5003982"),
  "productName" : "Rubberized Work Glove, Black",
  "tags" : [ "gardening" ] }
```

如果合并这些商品的tags，就可以取得名为testSet1的数组，如下所示：

```
testSet1 = ['tools']

db.products.aggregate([
  { $project:
    { productName: '$name',
      tags:1,
      setUnion: { $setUnion: ['$tags', testSet1] },
    }
  }
])
```

结果将包含如下所示的标签：

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"),
  "productName" : "Extra Large Wheel Barrow",
  "tags" : [ "tools", "gardening", "soil" ],
  "setUnion" : [ "gardening", "tools", "soil" ] }
```

工具、花园、土壤组合

```

}
{
  "_id" : ObjectId("4c4b1476238d3b4dd5003982"),
  "productName" : "Rubberized Work Glove, Black",
  "tags" : ["gardening"],
  "setUnion" : ["tools", "gardening"]
}

```

我们已经操作了各种不同的文档重塑函数，但是还有一个类别没有讲到：不出名的小众类别，我们把不属于前面类别的函数放到这里。

6.4.6 其他函数

最后一个分组混合了其他的函数，如表6.8所示。这些函数执行不同的功能，所以我们一起来介绍。`$meta`函数与文本搜索有关系，本章就不做介绍了。我们可以在第9章里学习更多关于文本搜索的内容。

表 6.8 其他函数

其他函数	
<code>\$meta</code>	文本搜索。参考第 9 章
<code>\$size</code>	返回数组大小
<code>\$map</code>	对数组的每个成员应用表达式
<code>\$let</code>	定义表达式内使用的变量
<code>\$literal</code>	返回表达式的值，而不评估它

`$size` 函数返回数组的大小。这个函数非常有用，例如，可用于判断数组是否包含某个元素或者是否为空。`$literal`函数允许我们避免初始化学段值为0、1、\$的问题。

`$let`函数允许我们使用临时变量，而不需要使用`$project`步骤。这个函数当我们要执行一系列复杂函数或者计算的时候非常有用。

`$map`函数允许我们来处理数组，并通过数组的元素执行函数来生成一个新的数组。`$map`在我们想重塑数组，但是又不想使用`$unwind`时非常有用。

本节学习完重塑文档的知识后，接下来将要研究一些与性能相关的问题。

6.5 理解聚合管道性能

Understanding aggregation pipeline performance

本节里我们将会学习如何改善聚合管道的性能，理解为什么管道性能会下降，也会学习如何突破中间或者最后输出结果的大小限制、从MongoDB 2.6开始删除的约束。

以下是主要影响聚合管道性能的关键点：

- 尽早管道里尝试减少文档的数量和大小。
- 索引只能用于\$match和\$sort操作，而且可以大大加速查询。
- 在管道使用\$match和\$sort之外的操作符后不能使用索引。
- 如果使用分片（分片存储大数据集合），则\$match和\$project会在单独的片上执行。一旦使用了其他操作符，其余的管道将会在主要片上执行。

本书全部内容都会鼓励大家尽可能多地使用索引。在第8章索引与查询优化中，我们会详细介绍这个主题。4个关键性能点中有2个都提到了索引，所以希望大家能意识到：索引可以大大加快大集合选择性查询和排序。

有时候，特别是当使用聚合框架的时候要处理大量的数据，此时索引不是恰当的方式。例如，6.2.2小节中通过年份和月份来计算销售数据时。处理大量的数据当然很好，但是用户不愿意长时间等待网页返回结果。当我们必须统计数据——例如，在网页上——通常可以提前生成数据并且使用\$out存储到集合里。

我们通过聚合框架的explain()函数来研究一下如何判断查询是否使用了索引机制。

6.5.1 聚合管道选项

直到现在，我们也只介绍了当传递数组给管道操作时调用aggregate()函数。从MongoDB 2.6开始，我们可以为aggregate()传递第二个参数来指定聚合调用。选项参数如下：

- explain()——运行管道并且只返回管道处理详细信息。
- allowDiskUse——使用磁盘存储数据。
- cursor——指定初始批处理的大小。

使用如下格式传递这些选项参数：

```
db.collection.aggregate(pipeline,additionalOptions)
```

pipeline就是之前例子里我们看过的管道操作的数组，additionalOptions是一个可选JSON对象，可以传递给aggregate()函数。additionalOptions的参数如下：

```
{explain:true, allowDiskUse:true, cursor: {batchSize: n} }
```

从explain()函数开始，我们来详细看看每个参数选项。

6.5.2 聚合框架的 explain() 函数

MongoDB explain()函数与SQL里的EXPLAIN函数类似，描述查询路径，允许开发者通过确定使用的索引来诊断慢操作。我们在第2章第一次讨论find()查询时介绍了explain()函数。列表6.2是复制列表2.2中的代码，用来说明如何使用索引来改善find()查询函数的性能。

列表6.2 索引查询的explain()函数输出

```
> db.numbers.find({num: {"$gt": 19995 }}).explain("executionStats")
```

```
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "tutorial.numbers",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "num" : {
        "$gt" : 19995
      }
    }
  },
  "winningPlan" : {
    "stage" : "FETCH",
    "inputStage" : {
      "stage" : "IXSCAN",
      "keyPattern" : {
        "num" : 1
      },
      "indexName" : "num_1",
      "isMultiKey" : false,
      "direction" : "forward",
      "indexBounds" : {
        "num" : [
          "(19995.0, inf.0]"
        ]
      }
    }
  },
  "rejectedPlans" : [ ]
},
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 4,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 4,
    "totalDocsExamined" : 4,
    "executionStages" : {
      "stage" : "FETCH",
      "nReturned" : 4,
      "executionTimeMillisEstimate" : 0,
      "works" : 5,
      "advanced" : 4,
      "needTime" : 0,
      "needFetch" : 0,
      "saveState" : 0,
      "restoreState" : 0,
      "isEOF" : 1,
      "invalidates" : 0,
      "docsExamined" : 4,
      "alreadyHasObj" : 0,
      "inputStage" : {
        "stage" : "IXSCAN",
        "nReturned" : 4,
        "executionTimeMillisEstimate" : 0,
        "works" : 4,
```

使用 num_1

返回 4 个
文档

只扫描 4 个文档

更快


```

    "advanced" : 4,
    "needTime" : 0,
    "needFetch" : 0,
    "saveState" : 0,
    "restoreState" : 0,
    "isEOF" : 1,
    "invalidates" : 0,
    "keyPattern" : {
      "num" : 1
    },
    "indexName" : "num_1",
    "isMultiKey" : false,
    "direction" : "forward",
    "indexBounds" : {
      "num" : [
        "(19995.0, inf.0]"
      ]
    },
    "keysExamined" : 4,
    "dupsTested" : 0,
    "dupsDropped" : 0,
    "seenInvalidated" : 0,
    "matchTested" : 0
  }
},
"serverInfo" : {
  "host" : "rMacBook.local",
  "port" : 27017,
  "version" : "3.0.6",
  "gitVersion" : "nogitversion"
},
"ok" : 1
}

```

使用 num_1 索引

聚合框架里使用explain()函数与find()查询里使用explain()函数不同,但是提供了类似的功能。正如我们期望的,对于聚合管道,我们会看到管道里每个操作的输出信息,因为管道里的每一步都是对下一步的调用(参见列表6.3)。

列表6.3 聚合函数的explain()输出。

```

> countsByRating = db.reviews.aggregate([
... {$match : {'product_id': product['_id']}},
... {$group : { _id:'$rating',
... count:{$sum:1}}}
... ],{explain:true})
{
  "stages" : [
    {
      "$cursor" : {
        "query" : {
          "product_id" : ObjectId("4c4b1476238d3b4dd5003981")
        },
        "fields" : {
          "rating" : 1,
          "_id" : 0
        }
      }
    }
  ]
}

```

优先匹配\$match

Explain 参数为 true

```

    },
    "plan" : {
      "cursor" : "BtreeCursor ",
      "isMultiKey" : false,
      "scanAndOrder" : false,
      "indexBounds" : {
        "product_id" : [
          [
            ObjectId("4c4b1476238d3b4dd5003981"),
            ObjectId("4c4b1476238d3b4dd5003981")
          ]
        ]
      },
      "allPlans" : [
        :
      ]
    }
  },
  "$group" : {
    "_id" : "$rating",
    "count" : {
      "$sum" : {
        "$const" : 1
      }
    }
  }
},
"ok" : 1
}

```

使用 BTreeCursor, 基于索引的光标

设置单个文档的范围

虽然这里显示的信息没有列表6.2中显示的find().explain()结果详细，但是它仍然提供一些关键的信息。例如，它显示了某个索引是否被显示，以及索引扫描的范围。这可以告诉我们哪个索引能够限制查询。

聚合框架的 explain()工作过程

explain() 是 MongoDB 2.6 新增的函数。因为之前版本缺少类似 find().explain() 的功能，所以新增的 explain() 函数可以提供类似的功能。正如 MongoDB 在线文档 <http://docs.mongodb.org/manual/reference/method/db.collection.aggregate/#example-aggregate-method-explain-option> 介绍的。此函数的输出结果是给人而不是给机器阅读的，输出结果的格式在不同的版本中可能有变化。如果结果与 find().explain() 一样也不要感到惊讶。find().explain() 函数在 MongoDB 3.0 已经进行了更大改进，包含了比 MongoDB 2.6 版本里 find().explain() 函数更多的输出信息，而且可以支持三种模式：queryPlanner、executionStats 和 allPlansExecution。

现在来看看另外一个解决之前限制处理数据大小的参数。

正如你已经知道的，`explain()` 函数输出结果可能与使用的MongoDB服务器有关。

6.5.3 allowDiskUse 选项

如果我们处理超大型集合数据，就会看到下面类似的错误：

```
assert: command failed: {
  "errmsg" : "exception: Exceeded memory limit for $group,
  but didn't allow external sort. Pass allowDiskUse:true to opt in.",
  "code" : 16945,
  "ok" : 0
} : aggregate failed
```

更令人沮丧的是，这个错误可能在很久之后发生，可能已经处理了几百万个文档，但是还是失败了。

发生这种情况的原因是，管道返回了超过MongoDB RAM内存限制的100MB数据。修复错误很简单，正如错误信息里提示的：设置`allowDiskUse:true`参数就可以了。

我们来看根据月份统计销售数据的例子。因为处理的销售数据量非常大，所以管道需要这个参数：

```
db.orders.aggregate([
  {$match: {purchase_data: {$gte: new Date(2010, 0, 1)}}}, ← 优先匹配$match
  {$group: {
    _id: {year : {$year : '$purchase_data'},
      month: {$month : '$purchase_data'}},
    count: {$sum:1},
    total: {$sum: '$sub_total'}},
  {$sort: {_id:-1}}
], {allowDiskUse:true}); ← 允许MongoDB使用磁盘存储
```

通常来说，使用`allowDiskUse`参数可能降低管道的性能，所以只推荐在需要的时候使用。正如前面提到的，我们应该尝试限制管道处理文档的数量和大小，使用`$match`选择处理的文档，使用`$project`选择要处理的字段。如果运行大数据的管道遇到这种情况，就使用这个参数会确保安全。

现在，我们来看下聚合管道里的最后一个参数：`: cursor`。

6.5.4 聚合光标选项

在MongoDB 2.6之前，管道结果的限制是单个文档16 MB。从2.6版本开始，通过Mongo shell访问MongoDB默认返回的是光标。但是，如果从程序里运行管道，为了避免程序崩溃而默认限制不变，则仍然限制返回文档大小是16MB。在程序里，我们可以通过如下的代码来访问数据库，返回光标结果：

```
countsByRating = db.reviews.aggregate([
  {$match: {'product_id': product['_id']}},
  {$group: {'_id': '$rating',
    count: {$sum: 1}}}
], {cursor: {}})
```

返回一个光标

聚合管道返回的光标支持如下调用：

- `cursor.hasNext()`——确定结果集是否包含下一个元素。
- `cursor.next()`——返回结果集的下一个文档。
- `cursor.toArray()`——以数组返回结果。
- `cursor.forEach()`——遍历结果集的每一行。
- `cursor.map()`——遍历结果集的每一行，返回一个结果数组。
- `cursor.itcount()`——返回结果数量（仅作测试）。
- `cursor.pretty()`——显示格式化结果的数组。

记住，光标(cursor)允许我们处理大规模数据流(stream)。它允许我们在返回少量文档结果的时候处理大的结果集，因此可以减少一次性处理数据所需的内存。此外，如果只需要一些文件，光标也可以限制服务端返回的文档数量。使用`toArray()`和`pretty()`就没有这些优势，结果会立即读入内存中。

类似地，`itcount()`会读取所有文档，并且全部发送给客户端，但是通常会抛弃结果，只返回数量。如果程序只需要数量，就可以使用`$group`管道操作符来计算输出文档的数量，而不需要发送每个文档给程序——这是更加高效的处理方式。

6.6 其他聚合功能

Other aggregation capabilities

虽然聚合管道是处理MongoDB聚合查询数据的首选方式，但也还有一些替代方式。有些非常简单，比如`.count()`函数。还有一个更复杂的方式，就是早版本MongoDB的`map-reduce`函数。

让我们先从简单的替代方式开始。

6.6.1 .count() 和 .distinct()

我们在6.2.1一节里已经看到了`.count()`函数。下面是摘录的部分代码：

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})
reviews_count = db.reviews.count({'product_id': product['_id']})
```

统计商品的
评价

现在来看看使用`.distinct()`函数的例子。下面代码将返回一串快递订单的邮政编码信

息数组，且不重复：

```
db.orders.distinct('shipping_address.zip')
```

.distinct() 返回结果的大小限制为16 MB，也是MongoDB文档的最大限制。

接下来我们来看一下最早版本MongoDB提供的聚合功能：map-reduce。

6.6.2 map-reduce

map-reduces 是MongoDB提供灵活聚合功能的首次尝试。使用map-reduce，就可以使用JavaScript定义整个处理流程。这提供了很大的灵活性，但是比聚合框架性能要低得多^[1]。此外，编写map-reduce的过程十分复杂，而且比之前构建的聚合框架更加难以理解。我们来看看之前的map-reduce框架应用例子。

注意：map-reduce的详细解释可以参考谷歌最初的论文，关于map-reduce的编程模型地址为 <http://static.googleusercontent.com/media/research.google.com/en/us/archive/mapreduce-osdi04.pdf>。

在6.2.2小节，我们展示了聚合管道提供的销售统计信息：

```
db.orders.aggregate([
  {"$match": {"purchase_data":{"$gte" : new Date(2010, 0, 1)}}},
  {"$group": {
    "_id": {"year" : {"$year" : "$purchase_data"},
           "month" : {"$month" : "$purchase_data"}},
    "count": {"$sum":1},
    "total": {"$sum":"$sub_total"}},
  {"$sort": {"_id":-1}}]);
```

我们使用map-reduce来生成相似的结果。第一步，正如名字的含义，就是编写map函数。这个函数会用到集合和处理过程中的每个文档上，有两个目的：第一，定义分组操作的键；第二，打包索引需要计算的数据。要了解这个过程，就来详细看看下面的函数：

```
map = function() {
  var shipping_month = (this.purchase_data.getMonth()+1) +
    '-' + this.purchase_data.getFullYear();

  var tmpItems = 0;
  this.line_items.forEach(function(item) {
    tmpItems += item.quantity;
  });

  emit(shipping_month, {order_total: this.sub_total,
                        items_total: tmpItems});
};
```

^[1]虽然 MongoDB 里改善了 JavaScript 的性能，但是还有一些关键原因，map-reduce 比聚合框架慢的多。这些问题的介绍请参考 William Zola 在 StackOverflow 网站的回答 <http://stackoverflow.com/questions/12678631/mapreduce-performance-in-mongodb-2-2-2-4-and-2-6/12680165#12680165>。

首先要知道，在这种情况下变量`this`引用的是当前迭代的文档——订单。在函数的第一行，我们获取一个值来指定订单创建的月份。然后调用`emit()`，这是每个`map`函数必须调用的一个专门的方法。

`emit()`的第一个参数是分组的关键值，第二个参数是包含值要处理的文档。此时，我们通过月份分组，然后统计每个订单的总额和项目数量。

对应的`reduce`函数如下：

```
reduce = function(key, values) {  
  var result = { order_total: 0, items_total: 0 };  
  values.forEach(function(value) {  
    result.order_total += value.order_total;  
    result.items_total += value.items_total;  
  });  
  return ( result );  
};
```

它会为`reduce`函数传递一个键和一个数组。我们编写`reduce`函数的工作就是确保这些值按照期望的方式来进行聚合处理，然后返回单个字。因为`map-reduce`的迭代本性，`reduce`可能会被多次调用。我们的代码必须处理这种问题。此外，如果`map`函数只发出一个值，`reduce`函数就不会被调用。因此，`reduce`返回的数据结构必须与`map`函数返回的数据结构一样。

我们来详细看一下`map`和`reduce`函数的例子。

添加查询过滤并保存输出结果

shell的`map-reduce`方法需要一个`map`和一个`reduce`函数作为参数。这个例子添加了更多参数。第一个是查询过滤器，用于限制聚合处理过程中的文档数量，从2010年开始。第二个参数是输出结果集合的名字。

```
filter = {purchase_data: {$gte: new Date(2010, 0, 1)}};  
db.orders.mapReduce(map, reduce, {query: filter, out: 'totals'});
```

过程如图6.6所示，包含以下步骤：

- (1) `filter` 选择特定的订单数据。
- (2) `map` 生成一个键值对，通常一个输入一个输出，但是它也可以不生成或者生成多个结果。
- (3) `reduce` 传递一个`key`和`map`生成的值的数组给`reduce`函数，通常是每个`key`一个数组，但是可能会使用不同值的数组传递多次相同的`key`。

图6.6所示中的一个重点就是，如果`map`函数为一个键生成一个结果，那么`reduce`步骤就会跳过去不执行。这也是理解为什么不能修改`map`函数的输出结果的数据结构的关键点，这对于`reduce`步骤非常重要。



图 6.6 map-reduce 处理过程

在这个例子里，结果存储在名为totals的集合中，我们可以查询这个集合。下面的列表显示了对于此集合的查询结果。`_id`字段保存的是分组的key，年份和月份还有value字段引用的总计信息。

列表6.4 查询map-reduce输出集合

```

> db.totals.find()
{ "_id" : "11-2014", "value" : { "order_total" : 4897, "items_total" : 1 } }
{ "_id" : "4-2014", "value" : { "order_total" : 4897, "items_total" : 1 } }
{ "_id" : "8-2014", "value" : { "order_total" : 11093, "items_total" : 4 } }
  
```

这里的例子应该可以让你对MongoDB的聚合功能有实际的理解。把这个例子和聚合框架的处理过程比较，就会发现map-reduce不再是这种功能推荐的处理方法了。

但是在有些情况下，我们需要map-reduce 提供的JavaScript的灵活性。本书里就不做介绍了，大家可以在MongoDB 网站 <http://docs.mongodb.org/manual/core/map-reduce/> 找到 map-reduce的例子。

map-reduce——很好的首次尝试

第一届全球MongoDB技术大会于2014年在美国纽约举行，MongoDB数据库的工程师团队展示了不同配置情况下处理几太字节数据的对比测试结果。一个工程师介绍了使用聚合框架而不是map-reduce的测试情况。当问到这个问题时，那个工程师说，map-reduce不再是推荐的处理数据的方式，但是是个“很好的首次尝试”。

虽然map-reduce提供了JavaScript的灵活性，但是它限制了必须是单线程和解释性的模式。聚合框架（aggregation framework），换句话说，是作为原生C++和多线程模式执行的。虽然map-reduce没有被淘汰，但是未来的改进都会在聚合框架上进行。

6.7 总结

Summary

本章包含许多的知识点。`$group`操作符提供了聚合框架的关键功能：从多个文档生成单个文档数据的功能。还有`$unwind` 和 `$project`，聚合框架提供了生成最新统计数据的功能，或者离线处理大量数据并使用`$out`命令保存到新的集合中的功能。

查询和聚合组成了MongoDB接口的关键部分。所以，一旦我们阅读完本章，就可以测试查询和聚合功能了。如果你还不确定某个查询操作符的组合如何工作，那么MongoDB shell是很好的测试工具。所以，尝试实战一下聚合框架的某些关键功能，比如使用\$match操作符选择文档，或者使用\$project操作符重建文档。当然还有使用\$group进行分组和统计信息。

从现在开始，我们将使用MongoDB的统一查询功能，而且接下来的一章会是很好的实战集合。我们会处理常见文档的CRUD操作：create、read、update、delete。因为在绝大部分更新操作里，查询都扮演了关键的角色，我们可以期待更多查询的详细知识。我们也会学习如何更新文档，特别是高并发大容量更新的数据库，这比大家熟悉的关系型数据库需要更多的功能。

这是一个简单的例子：

```
use test;
insert({_id: "1", name: "John", age: 25});
insert({_id: "2", name: "Mary", age: 30});
insert({_id: "3", name: "Frank", age: 35});
insert({_id: "4", name: "Alice", age: 40});
insert({_id: "5", name: "Bob", age: 45});
```

从MongoDB的文档类型来看，文档是JSON格式，所以文档的查询和更新操作是JSON格式。MongoDB的查询和更新操作是JSON格式，所以文档的查询和更新操作是JSON格式。MongoDB的查询和更新操作是JSON格式，所以文档的查询和更新操作是JSON格式。

MongoDB的查询和更新操作是JSON格式，所以文档的查询和更新操作是JSON格式。MongoDB的查询和更新操作是JSON格式，所以文档的查询和更新操作是JSON格式。MongoDB的查询和更新操作是JSON格式，所以文档的查询和更新操作是JSON格式。

本章里的大部分内容都使用MongoDB shell来操作，所以我们会使用MongoDB shell来操作。本章里的大部分内容都使用MongoDB shell来操作，所以我们会使用MongoDB shell来操作。本章里的大部分内容都使用MongoDB shell来操作，所以我们会使用MongoDB shell来操作。

```
use test;
var id = "1";
var doc = {name: "John", age: 25};
doc['_id'] = id;
print(doc);
print("inserted");
```

更新、原子操作和删除

Updates, atomic operations and deletes

本章内容

- 更新文档
- 自动处理文档
- 复杂更新的实际例子
- 使用更新操作符
- 删除文档

更新就是写入数据到现有的文档中。高效的做法需要完整理解文档结构的类型以及MongoDB支持的查询表达式。在过去的两章里我们已经学习了电商的数据模型，明白了schema数据模型设计和查询方式。在我们的更新学习中我们将会使用这些知识。

特别是，我们将会深入讲解为什么我们要使用非范式来设计类型的层次结构，以及MongoDB如何更新，让此数据结构看起来合理。

我们将会看一下库存管理，并且解决其中的一些并发问题。大家会学习一些新的更新操作符，包括一些原子更新的技巧，试验findAndModify命令。在这种情况下，MongoDB的原子性指的是查找并更新一个文档，确保不会被其他线程干扰。在许多例子之后，会有一节内容专门总结更新操作符，它会扩展所有的例子来允许我们通过参数控制如何更新数据。我们还会讨论如何删除MongoDB中的数据，最后来深入讲解一些MongoDB高并发和优化的知识。

本章里的绝大部分内容都使用JavaScript shell编写。本节里我们会讨论原子性文档处理工作，可能到时候需要许多应用级别的逻辑代码实现，所以我们将使用Ruby来编写。

在本章结束时，你会看到完整的MongoDB CRUD操作，并且可以熟练掌握最大化利用MongoDB数据库接口和数据模型优势的设计方法。

7.1 文档更新概要

A brief tour of document updates

更新MongoDB的数据库有两种实现方式。我们既可以完整替换现有的文档，也可以使用更新操作符来修改文档里的某个字段。在更详细的例子开始之前，我们先从简单的demo开始介绍这两种方式。然后介绍一下选择其中一种最优方式的理由。

大家回忆一下第4章里的user文档。这个文档包括用户的姓名、email邮箱，还有快递地址。

这是一个简单的例子：

```
{
  _id: ObjectId("4c4b1476238d3b4dd5000001"),
  username: "kbanker",
  email: "kylebanker@gmail.com",
  first_name: "Kyle",
  last_name: "Banker",
  hashed_password: "bd1cfa194c3a603e7186780824b04419",
  addresses: [
    {
      name: "work",
      street: "1 E. 23rd Street",
      city: "New York",
      state: "NY",
      zip: 10010
    }
  ]
}
```

毫无疑问，我们会经常更新电子邮件地址，所以现在就从它开始做吧。

请注意下，ObjectId的值可能不太一样，但确保使用有效的值。如果需要，就手动添加文档，这样可以帮助熟悉本章的命令。或者也可以使用下面的方法来查询有效的文档，获取它的ObjectId，在其他地方使用：

```
doc = db.users.findOne({username: "kbanker"})
user_id = doc._id
```

7.1.1 通过替换修改

要替换文档，首先要查询文档，在客户端修改，然后修改文档。下面是在JavaScript shell里使用命令的代码：

```
user_id = ObjectId("4c4b1476238d3b4dd5003981")
doc = db.users.findOne({_id: user_id})
doc['email'] = 'mongodb-user@mongodb.com'
print('updating ' + user_id)
db.users.update({_id: user_id}, doc)
```

有了用户_id, 首先可以查询到文档数据, 接下来可以本地修改文档。这种情况下, 修改email字段, 然后传递修改过的文档给update方法。最后一行意思是“使用给定的_id在users集合里查找文档”。要记住的是, 更新操作会取代整个文档, 这也就是要先获取文档数据的原因。如果多个用户更新同一个文档, 则只会保存最后更新的数据。

7.1.2 通过操作符修改

上面展示了如何通过替换文档修改数据, 现在来看看如何通过操作符来修改数据:

```
user_id = ObjectId("4c4b1476238d3b4dd5000001")
db.users.update({'_id': user_id},
                {'$set': {'email': 'mongodb-user2@mongodb.com'}})
```

这个例子使用的\$set是几个特殊的更新操作符之一, 用于修改单个文档中的email地址。这种情况下, 更新请求更具目标性: 找到文档, 然后修改email地址为mongodbuser2@mongodb.com。

Syntax note 语法提示: 更新与查询

MongoDB的新手区分更新和查询语法可能有点困难。目标更新通常使用更新操作符, 而且操作符通常使用动词构造 (set、push等)。以\$addToSet为例:

```
db.products.update({}, {'$addToSet': {'tags': 'Green'}})
```

如果为update添加查询选择器, 注意查询选择器语义上是类似的 (小于、等于等), 跟随在查询字段名字的后面(这个例子中的price):

```
db.products.update({'price': {'$lte': 10}},
                  {'$addToSet': {'tags': 'cheap'}})
```

最后一个查询例子只更新价格小于等于10的数据, 然后添加“cheap (便宜的)”标签。

更新操作符使用前缀表示符, 而查询操作符使用中缀表示符, 这意味着\$addToSet在更新操作符里是第一个, 而\$lte在查询操作符之内。

7.1.3 比较两个方法

另外一个例子怎么样? 这次我们要增加商品的评价。以下是使用替换方法实现此功能需求的:

```
product_id = ObjectId("4c4b1476238d3b4dd5003982")
doc = db.products.findOne({'_id': product_id})
doc['total_reviews'] += 1 // add 1 to the value in total_reviews
db.products.update({'_id': product_id}, doc)
```

这里是目标方法：

```
db.products.update({_id: product_id}, {$inc: {total_reviews: 1}})
```

替换方法也和前面一样，从服务器获取用户文档→修改→然后重新发回给服务器。这些代码与之前更新用户email地址的代码类似。与之相反，目标更新方法使用了不同的操作符\$inc来增加total_reviews字段的值。

7.1.4 决定：替换与操作符

既然我们已经看了许多更新的实战例子，能否给出一个选择更优方法的理由呢？哪个方法更直观？为什么某一个比另一个的性能更好？当多线程同时更新的时候会有什么问题？彼此之间是否隔离？

替换是更通用的做法。想象一下通过应用HTML表单来更新用户user数据。使用文档替换时，数据从表单提交，一旦验证，就可以传递给MongoDB；不管哪个字段被更新，代码执行的更新是相同的。例如，要构建一个MongoDB对象的映射器来支持更新，就可通过替换实现更新，这可能默认是合理的^[1]。

但是目标更新通常可以获得更好的性能，因为，不需要往返服务器来获取并修改文档数据。

而且，更重要的是文档更新通常很小。如果通过替换更新，每个文档平均200KB大小，则每次更新要接受和发送200 KB数据！

记住第5章我们讲过的，使用投影来获取文档的部分字段。如果要替换文档而不丢失数据，这并非理想的选择。与之前的例子使用\$set和\$push更新的方法正好相反，不论文档原始的大小，这些文档更新可能每次小于100B。因此，频繁使用目标更新意味着可以在序列化和传输数据上花费更少的时间。

此外，目标操作允许原子更新文档。例如，如果通过替换更新增加计数器，就可能会出现问題。如果读/写之间的时间发生变化修改呢？唯一方式是使用乐观锁（optimistic locking）来实现原子更新。使用目标更新，可以使用\$inc来原子性修改计数器。这意味着，即使大并发更新，每个\$inc都会隔离操作，要么成功要么失败^[2]。

^[1]绝大部分 MongoDB 对象映射器都使用了这个策略，容易理解其原因。如果用户可以建模任意复杂的实体，然后通过替换来更新会比使用特定的更新操作符来更新简单得多。

^[2]MongoDB 文档使用了原子更新这个词语来表示目标更新。这个新的术语是为了澄清 atomic 一词。事实上，所有发送给服务器的更新都是原子性的，基于每个文档进行隔离。更新操作符被原子性调用，因为这可以让查询和更新操作在单个操作内完成。

乐观锁

乐观锁 (optimistic locking) 或者乐观并发控制 (optimistic concurrency control), 是一种确保干净更新数据但是不需要锁定数据的技术。理解这个概念最好的例子就是wiki。可能同时有多个用户同时更新wiki页面, 但是又绝不希望有用户更新已经过时的页面信息。因此, 可以使用乐观锁协议。当用户尝试保存修改的时候, 可以包含一个尝试更新的时间戳。如果时间戳比最新保存的版本时间旧, 就不允许更新。如果没有用户保存任何编辑页面, 就允许更新。这个策略允许多个用户同时编辑, 它比另外一种需要用户锁定页面的并发策略更好。

使用悲观锁 (pessimistic locking), 记录会在事务里第一次访问时被锁定, 直到事务结束, 在此期间无法进行其他事务访问。

现在既然已经掌握了可用的更新方法, 那么我们就可以来学习下一节介绍的策略了。会返回到电商数据模型, 回答一些关于生产环境下操作数据更加困难的问题。

7.2 电商数据模型更新

E-commerce updates

更新MongoDB文档字段的例子很容易实现, 但是对产品数据模型和真实的应用程序, 复杂度会大大增加, 字段的更新可能无法用一行代码实现。

下面的小节里, 我们将会使用前两章提供的电商数据模型来实战练习更新操作。将会发现特定的更新非常直观, 没有这么复杂。从总体上讲, 我们会更好地理解第4章介绍的schema数据定义架构以及MongoDB更新语言的功能和局限性。

7.2.1 商品和目录

这里我们会看几个针对更新的例子, 首先我们来看一下如何计算商品的平均评分, 然后更复杂的任务是维护类别层次。

平均商品评分

商品数据模型适用于大量更新的策略。假设管理员提供了编辑商品信息的接口, 那么最简单的更新办法就是获取商品文档信息, 然后与用户的编辑信息合并, 使用替换策略更新文档。有时候我们可能只需要更新几个字段, 而此时适用于目标更新 (targeted update) 方式, 比如更新商品的评分。因为用户需要对商品评价进行排序, 所以需要在商品里存储评分的值, 当用户评分或者删除评分时再更新这个值。

这里是使用JavaScript更新操作方式的代码之一：

```
product_id = ObjectId("4c4b1476238d3b4dd5003981")
count = 0
total = 0
db.reviews.find({product_id: product_id}, {rating: 4}).forEach(
  function(review) {
    total += review.rating
    count++
  })
average = total / count
db.products.update({'_id: product_id',
  {$set: {total_reviews: count, average_review: average}})
```

这段代码从每个商品的评价中计算商品评价，然后计算出平均评分。我们也可以迭代计算所有的评分。这需要调用另外的函数count。有了总的评分和平均评分，代码就可以使用\$set来进行目标更新操作了。

如果不想硬编码ObjectId，可以使用如下代码来找到某个特定的ObjectId，然后再使用它：

```
product_id = db.products.findOne({'sku: '9092'}, {'_id': 1})
```

对性能敏感的用户，可能害怕每次更新重新聚合计算商品的评价。这个很大程度上取决于读/写的比例，其实查看商品评价的人数会远远超过更新的人数，所以当写入的时候重新计算是必要的。这里提供的方法虽然保守，但是绝大部分情况下还可以接受，当然其他策略也是可能的。例如，我们也可以在商品文档里存储额外的字段来保存评分信息，这样可以逐步计算平均评分。在插入新的评价后，我们可以先手动查询当前总的评分和平均评分，然后计算新的平均值，进而使用选择器更新文档。代码如下：

```
db.products.update({'_id: product_id',
  {
    $set: {
      average_review: average,
      ratings_total: total
    },
    $inc: {
      total_reviews: 1
    }
  })
```

这个例子使用了\$inc操作符，它可以基于给定的值进行增加运算——这里是每次加1。

只有根据系统的测试标准，使用代表性数据测试，才能判断这些做法是否合适。这个例子展示了MongoDB提供的一个有效方式。应用需求将会帮助你来确定哪个方式是最佳的。

类别层次

大部分数据库都没有好的办法表示类别层级。MongoDB也不例外，虽然它的文档结构可以一定程度上解决这些问题。文档鼓励读优化策略，因为每个类别可能都包含父类别的信息，这些都属于非范式设计。这里有个变态的要求就是保持父类别列表的更新。我们来看一下具体

如何实现这个需求。

首先，需要一个通用的更新祖先级别列表的方法，对于任意的类别都适用。这里是一个可行的解决方案：

```
var generate_ancestors = function(_id, parent_id) {
  ancestor_list = []
  var cursor = db.categories.find({_id: parent_id})
  while(cursor.size() > 0) {
    parent = cursor.next()
    ancestor_list.push(parent)
    parent_id = parent.parent_id
    cursor = db.categories.find({_id: parent_id})
  }
  db.categories.update({_id: _id}, {$set: {ancestors: ancestor_list}})
}
```

这个方法通过回溯遍历类别层级，依次查询每个节点的父类ID字段parent_id，直到到达根节点(parent_id为null)。此时，它构建了一个顺序的类别列表，存储在ancestor_list数组里。最后，它使用\$set来更新类别的ancestors字段。

既然我们已经编写了基本的代码块，现在就来看看插入新类别的代码。

假设有如图7.1所示的简单类别结构。

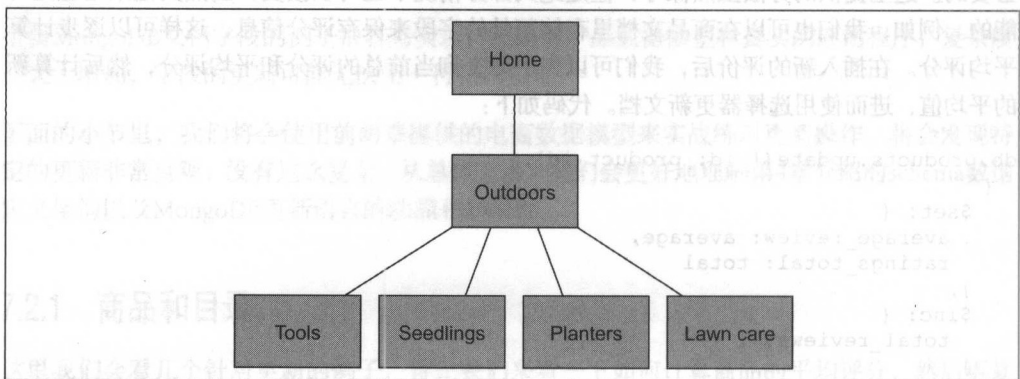


图 7.1 初始化类别层级

假设要添加一个新的类别Gardening（花园）放到Home（主页）类别下。首先，插入新文档，然后运行新方法来生成它的父类别：

```
parent_id = ObjectId("8b87fb1476238d3b4dd50003")
category = {
  parent_id: parent_id,
  slug: "gardening",
  name: "Gardening",
  description: "All gardening implements, tools, seeds, and soil."
}
```

```
db.categories.save(category)
generate_ancestors(category._id, parent_id)
```

注意：save()把创建的ID放入最初的文档里。这个ID用来调用generate_ancestors()方法。图7.2所示为更新后的树形结构。

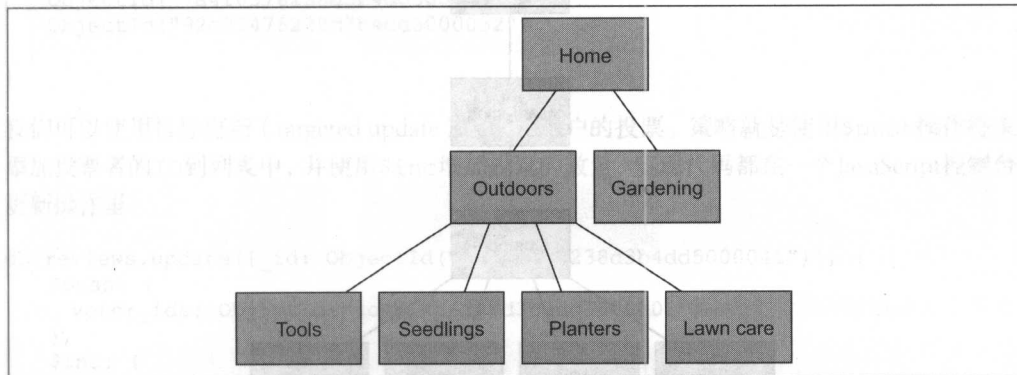


图 7.2 添加 Gardening 类

这非常简单。如果要把Outdoors类别插入到Gardening类下面呢？这就有点复杂了，因为它会修改父类别的列表。我们可以修改Outdoors类的parent_id为Gardening类的_id值就可以了。这样就不会太困难，因为我们有outdoors_id和gardening_id可用。

```
db.categories.update({'_id': outdoors_id}, {'$set': {'parent_id': gardening_id}})
```

因为我们已经高效地移动了Outdoors类别，所以Outdoors子类别的上级列表信息都无效了。可以使用Outdoors来查询所有的子类别，然后进行修改更新父类别的操作，这样就可维护类别层级的信息。

MongoDB强大的查询功能让这个需求变得小菜一碟，代码很简单：

```
db.categories.find({'ancestors.id': outdoors_id}).forEach(
    function(category) {
        generate_ancestors(category._id, outdoors_id)
    })
```

这就是如何更新类别的parent_id字段，新的类别结构如图7.3所示。

如果要更新类别名字，怎么办？如果要修改Outdoors为The Great Outdoors，就必须修改其他类别的父类别列表中所有出现Outdoors的地方。你可能会想，“这就是违反范式设计数据库带来的后果”！但是，如果知道下面的真相，就可能会感觉好点，因为不需要重新计算任意的上级类别列表来执行这个更新。下面是实现代码：

```
doc = db.categories.findOne({'_id': outdoors_id})
doc.name = "The Great Outdoors"
db.categories.update({'_id': outdoors_id}, doc)
db.categories.update(
```

```
{'ancestors._id': outdoors_id},
{$set: {'ancestors.$': doc}},
{multi: true})
```

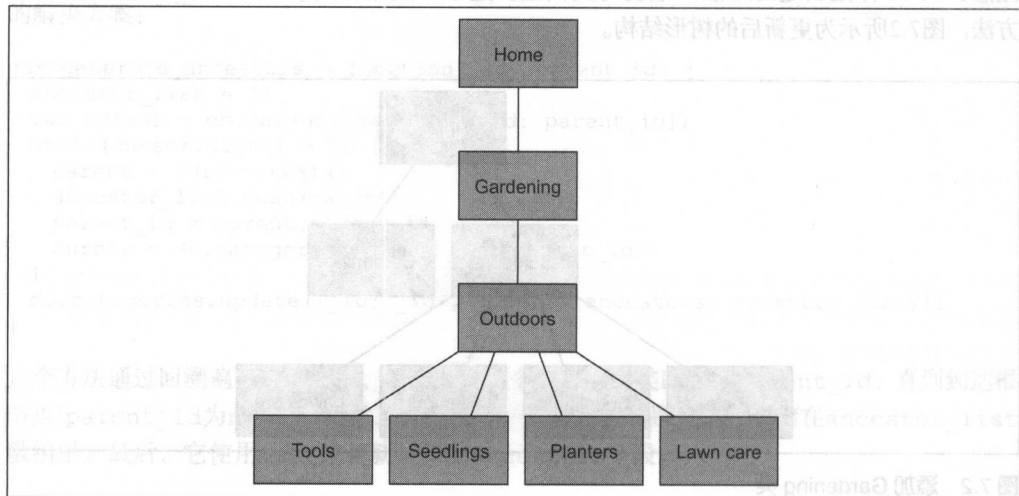


图 7.3 类别树的最终状态

首先要找到Outdoors文档，修改name字段，然后通过替换进行修改。现在，使用更新过的Outdoors文档来替换它出现过的所有列表中的上级类别信息。multi参数{multi: true}比较易于理解，它允许对所有匹配选择器的文档执行更新——如果没有{multi: true}，则只会更新第一个匹配的文档。这里我们想要更新每个包含Outdoors为上级类别的列表。

位置操作符更加微妙。假设我们不知道Outdoors类别出现在哪个类的父类列表中。这就需要动态定位Outdoors类别在列表中的位置。输入位置定位符。这个符号(\$ 在ancestors.\$中)使用自己来替代匹配选择器数组元素的索引，启用更新。

下面是这个技巧的又一个例子。假设想要修改用户的地址address信息(7.1节里的例子文档)，已经标记为“work”。我们可以使用如下的代码来解决这个问题：

```
db.users.update({
  _id: ObjectId("4c4b1476238d3b4dd5000001"),
  'addresses.name': 'work',
  {$set: {'addresses.$.street': '155 E 31st St.'}})
```

因为需要更新数组中的单个文档，所以希望保留位置操作符。通常，这个方法适用于更新类别层级信息。无论什么时候处理文档数组，我们都可以实现定位更新。

7.2.2 评价

并非所有的评价都一样，因此允许用户去投票。这些投票是原始的，可以表示那些评价是有帮助的。我们已经建模了评价，它们可以用来缓存有帮助的投票数量，并保存一个投票人的

ID列表。每个相关的评价文档的结构如下所示：

```
{
  helpful_votes: 3,
  voter_ids: [
    ObjectId("4c4b1476238d3b4dd5000041"),
    ObjectId("7a4f0376238d3b4dd5000003"),
    ObjectId("92c21476238d3b4dd5000032")
  ]
}
```

我们可以使用目标更新（targeted update）来记录用户的投票。策略就是使用\$push操作符来添加投票者的ID到列表中，并使用\$inc增加投票的数量，实现代码都在一个JavaScript控制台更新操作里：

```
db.reviews.update({_id: ObjectId("4c4b1476238d3b4dd5000041")}, {
  $push: {
    voter_ids: ObjectId("4c4b1476238d3b4dd5000001")
  },
  $inc: {
    helpful_votes: 1
  }
})
```

这个代码基本正确了，但是我们要确保只有没有投过票的用户才能投票，所以需要修改查询选择器，只有voter_ids数组不包含用户的ID才能投票。这个使用\$ne查询操作符很容易实现：

```
query_selector = {
  _id: ObjectId("4c4b1476238d3b4dd5000041"),
  voter_ids: {
    $ne: ObjectId("4c4b1476238d3b4dd5000001")
  }
}
db.reviews.update(query_selector, {
  $push: {
    voter_ids: ObjectId("4c4b1476238d3b4dd5000001")
  },
  $inc: {
    helpful_votes: 1
  }
})
```

这个例子专门演示了MongoDB强大的更新机制，以及它如何与面向文档的schema一起使用。投票，在这个例子里是原子性和高效的。更新是原子性的，因为查询和修改都在单个操作里完成。原子性确保，即使在高并发环境里，它都可以确保每个用户不会投票超过1次。这个高效性依赖于在同一个请求中对于投票资格的检验，以及更新计数器和投票者列表。

现在，如果你最终使用这个技术来记录投票，特别重要的是其他针对这个评价的更新操作也要是目标性的——替换更新可能导致数据的不一致性。想象一下，用户更新了它们评价的内容，而这次更新是通过替换完成的。当通过替换更新时，首先要查询要更新的文档数据。但

是在查询和更新的操作之间，可能有不同的用户来投票修改这个评价文档的数据。这叫做竞争条件（race condition）。这个事件导致的结果如图7.4所示。

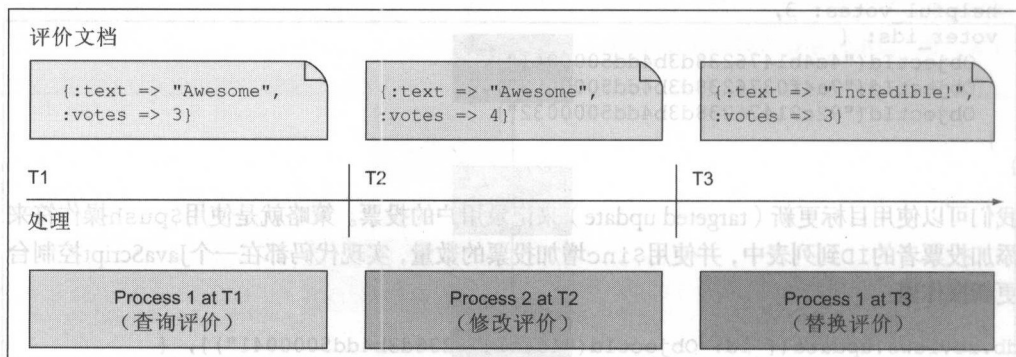


图 7.4 当通过目标和替换更新来并发更新评价时，数据可能会丢失

很明显，这个发生在T3的文档替换会覆盖T2的更新数据。可以通过使用乐观锁技术来避免这种事情发生，但是这样做需要额外的应用代码来实现乐观锁，而且确保所有的更新都是目标性的，在这个例子中更容易实现。

7.2.3 订单

我们在评价里看到更新操作的原子性和高效性都可以应用到Orders订单。特别是，我们将要看到MongoDB调用需要使用目标更新来实现add_to_cart函数。这是个三步过程。首先，需要构造订单商品数组里的商品文档。其次，要使用目标更新，表明这是一个upsert——如果更新的文档不存在，就创建它（在下一节里会详细介绍upsert）。最后，如果订单不存在，upsert将会创建新的订单对象，在初始和后续添加到购物车^[1]的过程中实现无缝处理。

我们来构建一个例子文档，然后添加到购物车里：

```
cart_item = {
  _id: ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",
  name: "Extra Large Wheel Barrow",
  pricing: {
    retail: 5897,
    sale: 4897
  }
}
```

我们很可能通过查询products集合来构建这个文档，然后抽取需要的字段来保存到订单中。对于这个商品，_id、sku、slug、name、price字段应该足够了。接下来，使用参数{upsert:

^[1]我们使用了可以互换的词汇购物车（shopping cart）和订单（order），因为它们都使用了相同的文档表示。它们是由文档的state字段区分的（CART状态表示购物车）。

true}确保有个状态为“CART”的客户订单。

这个操作也会使用\$inc操作符增加订单的商品数量sub_total:

```
selector = {
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART'
}
update = {
  $inc: {
    sub_total: cart_item['pricing']['sale']
  }
}
db.orders.update(selector, update, {upsert: true})
```

通过初始化 upsert 来创建订单文档

为解析一下代码，我们分别构建了查询器和更新文档。更新文档通过商品价格增加了订单的总额。当然，用户第一次执行add_to_cart函数时，购物车是不存在的。这也是为什么我们使用upsert。用upsert将会创建由查询器定位的文档。因此初始化的upsert将会生成一个如下的订单文档：

```
{
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART',
  subtotal: 9794
}
```

然后执行order的更新操作把商品添加进来。如果商品不在订单上，就新增一个：

```
selector = {user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART',
  'line_items._id':
    {'$ne': cart_item._id}
}

update = {'$push': {'line_items': cart_item}}
db.orders.update(selector, update)
```

更新数量

接下来，我们使用另外一个目标更新来确保商品数量的正确性。我们需要这个更新操作来处理特殊的情况，比如用户点击“Add to Cart”时，商品已经添加到购物车里了。此时，之前的更新不会再添加新商品到购物车，但是需要调整订单里商品的数量：

```
selector = {
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART',
  'line_items._id': ObjectId("4c4b1476238d3b4dd5003981")
}
update = {
  $inc: {
```

```

    'line_items.$.quantity': 1
  }
}
db.orders.update(selector, update)

```

我们使用\$inc操作符来更新单个商品的数量。使用之前介绍的位置操作符\$update也非常方便。因此，在用户两次点击独轮车商品的Add to Cart按钮后，购物车的结果如下所示：

```

{
  user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  state: 'CART',
  line_items: [
    {
      _id: ObjectId("4c4b1476238d3b4dd5003981"),
      quantity: 2,
      slug: "wheel-barrow-9092",
      sku: "9092",
      name: "Extra Large Wheel Barrow",
      pricing: {
        retail: 5897,
        sale: 4897
      }
    }
  ],
  subtotal: 9794
}

```

应该有2辆独轮车在购物车里，商品数量应该正确地显示了2个。

要完整实现购物车还需要很多其他的操作。绝大部分操作，比如从购物车删除商品或者清空购物车时都可以使用一个或者多个目标实现。如果还不明显，接下来的内容会介绍每个查询操作符，让你理解得更清晰。实际的订单处理过程可以通过一系列状态和逻辑组合来进行处理。

我们将会在下一节里进行演示，那里我们会介绍文档的原子处理以及findAndModify命令。

7.3 原子文档处理

Atomic document processing

不想错过的一个工具就是MongoDB的findAndModify命令^[1]。

这个命令允许我们在同一个往返过程中原子更新文档并返回它。原子更新就是一个不会被其他更新中断或者与其他操作交互的操作。如果用户在我们找到这个文档后修改之前尝试修改此文档呢？这个查找不会成功。原子更新会阻止这个情况，所有其他操作必须等待原子更新完成才行。

^[1]这个命令会根据环境变化。shell 帮助类调用使用了驼峰命名法则 db.orders.findAndModify, Ruby 使用了下划线: find_and_modify。核心服务器会识别 findandmodify。如果要手动调用命令可以使用最终的格式。

每个MongoDB更新都是原子性的,但是与findAndModify不同的是它会自动返回文档给你。为什么这非常有用呢?因为当你要获取并更新一个文档(或更新并获取它)时,可能有另外一个MongoDB用户修改这个文档,虽然更新是原子性的,但也不可能知道我们更新文档的真实状态(更新前或者后),除非使用了findAndModify。其他选择是使用7.1节提到的乐观锁机制,这需要额外的应用逻辑来实现。

原子更新非常重要是因为它支持许多功能。例如,可以使用findAndModify来构建工作队列和状态机,然后使用这些原始语句来构建事务语义,这极大地扩展了MongoDB的应用范围。使用事务性特性,我们可以使用MongoDB开发完整的电商网站——不仅仅是商品内容,还包括结账机制以及库存管理。

为了演示,我们使用了两个findAndModify命令的例子。首先,我们看一下如何处理基本的购物车状态转换。然后我们会看一个关于库存管理的稍微复杂的例子。

7.3.1 订单状态转换

所有的状态转换包含2个部分:查询、确保有效的初始状态和一个影响状态修改的更新。我们跳过一些订单处理的步骤,假设用户要点击支付(Pay Now)按钮来授权交易。如果要在应用端同步授权信用卡支付,就需要以下4步:

- (1) 授权用户查看结算页面。
- (2) 在授权过程中购物车内容不变。
- (3) 授权过程中的任何错误都会导致购物车返回之前的状态。
- (4) 如果信用卡授权成功,支付信息会提交给订单,然后订单状态转换为PRE-SHIPPING(准备快递)。

我们使用了状态转换,如图7.5所示。

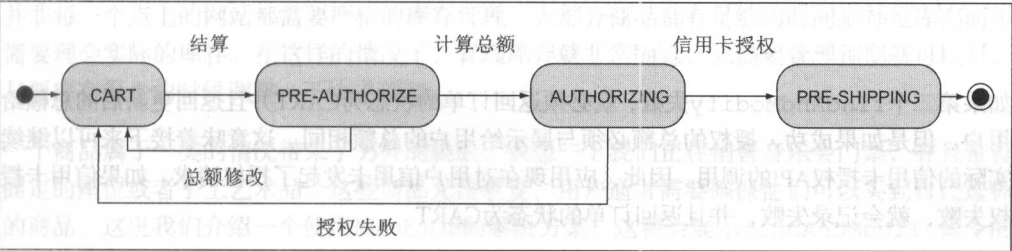


图 7.5 订单状态转换

准备结算订单

第一步要进入PRE-AUTHORIZE(预授权)状态。我们使用findAndModify来查找用户当前

的订单对象，并确保对象在CART状态：

```
newDoc = db.orders.findAndModify({
  query: {
    user_id: ObjectId("4c4b1476238d3b4dd5000001"),
    state: 'CART'
  },
  update: {
    $set: {
      state: 'PRE-AUTHORIZE'
    }
  },
  'new': true
})
```

如果成功，findAndModify会返回修改过的订单对象给newDoc^[1]。一旦订单是PRE-AUTHORIZE（预备授权）状态，用户就不能编辑购物车内容。这是因为所有的更新都必须确保CART购物车的状态。findAndModify非常有用，能让我们知道当修改状态为PRE-AUTHORIZE时文档的准确状态。如果另外一个线程也在尝试修改用户的结算流程，会发生什么问题？

检查订单和授权

现在，在预授权状态下我们获取了返回的订单对象，并且重新计算各个商铺的总和。一旦计算完成，就可以使用findAndModify把文档状态转换为AUTHORIZING。当然，新的总额必须与旧的总额匹配才行。以下就是使用findAndModify的例子代码：

```
oldDoc = db.orders.findAndModify({
  query: {
    user_id: ObjectId("4c4b1476238d3b4dd5000001"),
    total: 99000,
    state: "PRE-AUTHORIZE"
  },
  update: {
    '$set': {
      state: "AUTHORIZING"
    }
  }
})
```

如果第二个findAndModify失败，就必须返回订单的状态为CART并且返回更新后的总额给用户。但是如果成功，授权的总额必须与展示给用户的总额相同。这意味着接下来可以继续实际的信用卡授权API的调用。因此，应用现在对用户信用卡发起了授权请求。如果信用卡授权失败，就会记录失败，并且返回订单的状态为CART。

^[1]默认情况下，findAndModify命令会返回文档更新之前的状态。要返回修改后的文档，就必须制定'new': true 参数。正如这个例子一样。

完成订单

如果授权成功，就要把授权信息写入订单，然后转换到下一状态。下面的策略是调用 `findAndModify` 做了同样的工作。这里的例子使用文档来表示授权接受信息，它被附加到最初的订单中：

```
auth_doc = {
  ts: new Date(),
  cc: 3432003948293040,
  id: 2923838291029384483949348,
  gateway: "Authorize.net"
}
db.orders.findAndModify({
  query: {
    user_id: ObjectId("4c4b1476238d3b4dd5000001"),
    state: "AUTHORIZING"
  },
  update: {
    $set: {
      state: "PRE-SHIPPING",
      authorization: auth_doc
    }
  }
})
```

非常重要的一点就是要知道MongoDB的功能使用了这个事务性过程。它具备原子性修改任意文档的能力，还可以确保单一连接中读取的一致性。最后，文档结构自身允许这些操作适应MongoDB提供的单个文档的原子性。在这个例子中，这个结构允许我们把商品列表、价格、用户资料保存到单个文档里，确保我们只需要在单个文档上操作就可以了，提升了伸缩性。

这应该会让你印象深刻。同时它也可能让你好奇，MongoDB里能否支持多个对象（multi-object）的事务操作？

7.3.2 库存管理

并非每一个点上的网站都需要严格的库存管理。大部分商品都有足够的时间来补足库存而不需要理会实际的库存。在这样的情况下，管理库存就非常随意，只需要管理预期就可以了，只要库存很少的时候调整一下快递预期。

一个商品属于一类的情况带来了另外的挑战。设想一下我们正在销售音乐会门票，并且带有固定的座位或者手工艺品。这些商品无法重复，用户通常需要确保他们可以买到自己选择的商品。这里我们介绍一个使用MongoDB的解决方案。这将会展示 `findAndModify` 命令的性能，以及表明使用这个文档模型是非常明智的选择；也会展示如何在多个文档中实现事务性语义。虽然我们只会看到此过程中的一些关键的MongoDB调用，但是完整的 `InventoryFetcher` 类的代码也包含在本书中。

建模库存的最佳理解方式就是通过真实的商店例子。如果你去过花园商店，就可以看到和感

觉到实际的物理库存：铁锹、钉耙和剪刀放在货架上。如果选择了铁锹并放入购物车，那么顾客可选的铁锹数量就会少一把。作为推论，2个顾客不能同时选择相同的铁锹放到购物车里。我们可以使用这个简单的原则来建模。对于每个仓库中的物品，集合中都存储一个文档。如果有10把铁锹，在数据库里就有10个有关铁锹的文档。每个库存项目都会通过sku连接到一个商品上，每个项目可以有4个状态：AVAILABLE (0)，IN_CART (1)，PRE_ORDER (2)，PURCHASED (3)。

这里的代码插入铁锹、钉耙和剪刀作为可用的3个状态。例子代码使用Ruby编写，因此事务需要更多的逻辑，所以看看具体如何使用的例子会更有帮助：

```
3.times do
  $inventory.insert_one({:sku => 'shovel', :state => AVAILABLE})
  $inventory.insert_one({:sku => 'rake', :state => AVAILABLE})
  $inventory.insert_one({:sku => 'clippers', :state => AVAILABLE})
end
```

我们将会使用专门的库存提取的类来处理库存管理。首先我们来看一下这个类如何工作，然后会详细看一下它的实现代码。

库存提取器

用这个库存提取器可以向购物车里添加任意商品的集合。这里我们创建了一个新的订单对象和一个新的库存提取器。然后让提取器添加3把铁锹和一些剪刀到订单里，通过订单ID识别订单，用另外2个文档指定商品和数量，传递给方法add_to_cart。此提取器隐藏了操作的复杂性，它可以一次性修改2个集合：

```
$order_id = BSON::ObjectId('561297c5530a69dbc9000000')
$orders.insert_one({
  :id => $order_id,
  :username => 'kbanker',
  :item_ids => []
})

@fetcher = InventoryFetcher.new({
  :orders => $orders,
  :inventory => $inventory
})

@fetcher.add_to_cart(@order_id,
[
  {:sku => "shovel", :quantity => 3},
  {:sku => "clippers", :quantity => 1}
])

$orders.find({"_id" => $order_id}).each do |order|
  puts "\nHere's the order:"
  p order
end
```

如果添加项目到购物车里，add_to_cart方法将会抛出一个异常。如果成功，订单应该如下

所示:

```
{
  "_id" => BSON::ObjectId('4cdf3668238d3b6e3200000a'),
  "username" => "kbanker",
  "item_ids" => [
    BSON::ObjectId('4cdf3668238d3b6e32000001'),
    BSON::ObjectId('4cdf3668238d3b6e32000004'),
    BSON::ObjectId('4cdf3668238d3b6e32000007'),
    BSON::ObjectId('4cdf3668238d3b6e32000009')
  ]
}
```

每个物理库存项目的_id将会存储到单独的订单文档里。人们可以如下方式来查询每个项目:

```
puts "\nHere's each item:"
order['item_ids'].each do |item_id|
  item = @inventory.find({"_id" => item_id}).each do |myitem|
    p myitem
  end
end
```

单独看每个商品项目,可以看到每个都有一个状态1,对应的状态是IN_CART。应该注意到每个条目记录了状态修改的最后时间。我们可以在后面使用这个时间戳来过期已经长时间存储在购物车里的物品。例如,我们可能给用户15分钟来结算订单,购物车的商品可以存放15分钟:

```
{
  "_id" => BSON::ObjectId('4cdf3668238d3b6e32000001'),
  "sku"=>"shovel",
  "state"=>1,
  "ts"=>"Sun Nov 14 01:07:52 UTC 2010"
}
{
  "_id"=>BSON::ObjectId('4cdf3668238d3b6e32000004'),
  "sku"=>"shovel",
  "state"=>1,
  "ts"=>"Sun Nov 14 01:07:52 UTC 2010"
}
{
  "_id"=>BSON::ObjectId('4cdf3668238d3b6e32000007'),
  "sku"=>"shovel",
  "state"=>1,
  "ts"=>"Sun Nov 14 01:07:52 UTC 2010"
}
```

库存管理

如果InventoryFetcher的API合理,就应该对于如何实现库存管理有了初步的想法。不出意外,findAndModify命令会出现在核心代码中。InventoryFetcher的源码出现在本书的源代码中。我们不会在本书里列出每一行代码,但是我们将重点介绍其中的3个方法。

首先,当要传递列表项目给购物车时,fetcher获取器会尝试把每个商品的状态AVAILABLE转

换为IN_CART。如果任意一个操作失败（某一个添加失败），整个操作就会回滚。来看一下我们之前调用的方法add_to_cart:

```
def add_to_cart(order_id, *items)
  item_selectors = []
  items.each do |item|
    item[:quantity].times do
      item_selectors << {:sku => item[:sku]}
    end
  end
  transition_state(order_id, item_selectors,
    {:from => AVAILABLE, :to => IN_CART})
end
```

*items语法允许用户传递任意数量的对象，它被放置到一个名为items的数组里。这个方法工作不多，它把商品添加到购物车里，并且扩展数量以保证每个要添加到购物车的商品项目都有一个物理商品对应一个选择器。例如，这个文档表示我们要添加2把铁锹到购物车里：

```
{:sku => "shovel", :quantity => 2}
```

变成这个:

```
[{:sku => "shovel"}, {:sku => "shovel"}]
```

每个要添加到购物车的商品都有单独的查询选择器。因此，这个方法传递商品选择器的数组给transition_state方法。例如，之前的代码指定了状态应该从AVAILABLE转换到IN_CART:

```
def transition_state(order_id, selectors, opts={})
  items_transitioned = []
  begin # 使用begin/end块可以为选择器进行错误恢复
    query = selector.merge({:state => opts[:from]})
    physical_item = @inventory.find_and_modify({
      :query => query,
      :update => {
        '$set' => {
          :state => opts[:to], # 目标状态
          :ts => Time.now.utc # 获取当前客户端时间
        }
      }
    })
    if physical_item.nil?
      raise InventoryFetchFailure
    end

    items_transitioned << physical_item['_id'] # 项目加入数组中
    @orders.update_one({:_id => order_id}, {
      '$push' => {
        :item_ids => physical_item['_id']
      }
    })
  end # 结束循环
```

```

rescue Mongo::OperationFailure, InventoryFetchFailure
  rollback(order_id, items_transitioned, opts[:from], opts[:to])
  raise InventoryFetchFailure, "Failed to add #{selector[:sku]}"
end

return items_transitioned.size
end

```

要进行状态转换，每个选择器获取一个额外的条件`{:state => AVAILABLE}`，同时选择器发送给`findAndModify`，如果匹配，就修改商品时间戳和新状态。这个方法保存转换状态的项目，并且使用新的商品ID更新订单。

优雅的失败

如果`findAndModify`命令失败，并返回`nil`，就抛出一个`InventoryFetchFailure`异常。如果是因为网络问题失败，就抛出`Mongo::OperationFailure`异常。在两种情况下，都可以通过回滚目前的转换状态并抛出`InventoryFetchFailure`异常，它包含无法添加的商品SKU。我们也可以应用代码里优雅地处理这个异常。

现在剩下的全部工作就是检查回滚代码了：

```

def rollback(order_id, item_ids, old_state, new_state)
  @orders.update_one({"_id" => order_id},
    {"$pullAll" => {:item_ids => item_ids}})
  item_ids.each do |id|
    @inventory.find_one_and_update({
      :query => {
        "_id" => id,
        :state => new_state
      },
      {
        :update => {
          "$set" => {
            :state => old_state,
            :ts => Time.now.utc
          }
        }
      })
  end
end

```

我们使用`$pullAll`操作符来删除所有添加到订单`item_ids`数组的ID。然后迭代ID列表，转换每个商品的状态为旧的状态。`$pullAll`操作符与其他更新操作符一样会在7.4.2小节里详细介绍。

`transition_state`方法可以用作其他方法的基础，其他方法可以继续转换这些商品的状态。把它集成到订单转换系统中应该不困难，这里就留给读者作为练习作业了。

这实现代码里忽略的一个情况就是，库存商品无法恢复到它们的初始状态。如果Ruby驱动无

法与MongoDB通信，或者在完成之前过程运行了回滚挂起，这就会让购物车商品留在IN_CART状态，但是订单集合不会包含库存。此种情况下，管理交易变得困难。这些都会最终被修复，但是，通过购物车超时可以删除这些超时的商品。

你可能会怀疑这个系统作为生产环境的健壮性。这个问题不能轻易回答，除非了解更多的特殊性，但是可以肯定的是，MongoDB提供了足够的功能来确保一个可用的事务性交易解决方案。MongoDB从未打算支持多个集合的事务，但是它允许用户使用find_one_and_update和乐观并发控制来模拟这个行为。如果发现经常要管理事务，则最好考虑使用不同的schema或者数据库。并非所有的应用都适合MongoDB，但是如果仔细规划schema就可以避免这种事务的需求。

7.4 核心要点：MongoDB 更新与删除

Nuts and bolts: MongoDB updates and deletes

要理解MongoDB更新机制，就需要完整理解MongoDB的文档模型和查询语言，之前章节的例子就是最好的帮助资料。正如本章开始承诺的，我们将会深入谈到一些本质问题。这会涉及每个MongoDB更新接口的功能总结，我们也会介绍几个性能问题的注意事项。为了简洁起见，后面的例子我们大部分使用JavaScript代码编码。

7.4.1 更新类型与参数选项

正如我们之前例子展示的，MongoDB支持目标更新和替换更新。前者通过使用一个或多个操作符定义；后者通过使用文档来替换匹配查询器的文档实现。

注意，如果更新文档非常含糊，那么更新可能失败。这是使用MongoDB的常见问题，而且是非常容易犯的错误。这里，我们已经集合使用了更新操作符\$addToSet，使用了替换语义{name:"Pitchfork"}：

```
db.products.update_one({}, {name: "Pitchfork", $addToSet: {tags: 'cheap'}})
```

如果要修改文档的名字，就必须使用\$set操作符：

```
db.products.update_one({},  
{$set: {name: "Pitchfork"}, $addToSet: {tags: 'cheap'}})
```

多文档更新

默认情况下，只会更新匹配查询器的第一个文档。要更新所有的匹配文档，就需要显示指定多文档更新模式。在shell里，可以通过添加参数multi: true来实现。以下是如何在商品集合里添加cheap标签到所有的文档中：

```
db.products.update({}, {$addToSet: {tags: 'cheap'}}, {multi: true})
```

在文档级别更新是原子性的，这意味着一条更新10个文档的语句可能在更新3个文档后由于某些原因失败。应用程序必须根据自己的策略来处理这些失败。

使用Ruby驱动（以及其他驱动），可以使用与以下相似的方式来处理多个文档更新：

```
@products.update_one({},  
  {'$addToSet' => {'tags' => 'cheap'}},  
  {:multi => true})
```

UPSERTS

有个问题非常常见，就是当文档存在时更新，文档不存在时插入数据。我们可以使用upsert来处理这种常见的问题。如果查询选择器匹配，更新就正常执行。如果没有匹配的文档，就会插入新的文档。新文档的字段是查询选择器和目标更新文档的逻辑合并^[1]。

下面是shell使用upsert的简单例子，设置upsert: true参数来允许upsert：

```
db.products.update({slug: 'hammer'},  
  {$addToSet: {tags: 'cheap'}}, {upsert: true})
```

以下是等价的Ruby实现upsert的代码：

```
@products.update_one({'slug' => 'hammer'},  
  {'$addToSet' => {'tags' => 'cheap'}}, {:upsert => true})
```

正如你所期望的，upserts一次可以插入或者更新一个文档。当需要更新原子性和不确定文档是否存在时，upserts非常有用。对于特别的例子，参考7.2.3，它描述了添加商品到购物车的过程。

7.4.2 更新操作符

MongoDB支持一系列更新操作。这里我们介绍一下每个操作符的简单使用例子。

标准操作符

第一个操作符集合是最通用的，每个都可以用来处理任意数据类型。

\$inc

可以使用\$inc操作符来增加或者减少一个数值：

```
db.products.update({slug: "shovel"}, {$inc: {review_count: 1}})  
db.users.update({username: "moe"}, {$inc: {password_retries: -1}})
```

^[1]注意：upsert 无法与替换更新文档模式一起工作。

也可以使用\$inc来添加或者减去任意的数值：

```
db.readings.update({_id: 324}, {$inc: {temp: 2.7435}})
```

\$inc的高效就如其便捷性一样。因为它很少用于修改文档的大小，\$inc通常发生在磁盘上，因此只影响指定的值^[1]。

之前的观点只适用于MMAPv1存储引擎。WiredTiger存储引擎的工作方式不同，它使用先写事务日志的方式，与检查点一起使用以确保数据的一致性。

正如往购物车里添加商品的演示代码一样，\$inc与upserts一起使用。例如，可以修改之前的更新代码为upsert，如下所示：

```
db.readings.update({_id: 324}, {$inc: {temp: 2.7435}}, {upsert: true})
```

如果_id为324的文档不存在，就会创建一个新的文档，_id是324，temp的增加值为2.7435。

\$set 和 \$unset

如果要设置某个文档里的特定key值，就可以使用\$set。

我们可以为key设置任意有效的BSON类型值。这意味着下面的代码都是有效的：

```
db.readings.update({_id: 324}, {$set: {temp: 97.6}})
db.readings.update({_id: 325}, {$set: {temp: {f: 212, c: 100}}})
db.readings.update({_id: 326}, {$set: {temps: [97.6, 98.4, 99.1]}})
```

如果key键已经存在，它的值就会被重写；否则，创建新的key值。

\$unset会从文档中删除提供的key。以下是从文档里删除temp键的例子代码：

```
db.readings.update({_id: 324}, {$unset: {temp: 1}})
```

我们也可以在嵌入式文档和数组里使用\$unset。两种情况下，都使用了原点符号来指定内部对象。如果集合里存在以下2个文档：

```
{_id: 325, 'temp': {f: 212, c: 100}}
{_id: 326, temps: [97.6, 98.4, 99.1]}
```

就可以在第一个文档里删除华氏摄氏度（Fahrenheit）以及第二个文档里的“0(zeroth)”元素，如下所示：

```
db.readings.update({_id: 325}, {$unset: {'temp.f': 1}})
db.readings.update({_id: 326}, {$pop: {temps: -1}})
```

原点访问子文档或者数组元素也可以用到\$set中。

^[1]当数值类型修改时根据这个规则会抛出异常。如果用\$inc把32位整数转换为64位整数，那么整个BSON文档都会被替换重写。

数组中使用\$unset

注意，在单个数组元素上使用\$unset可能无法像我们期望的那样准确地工作。它仅仅会设置元素的值为null而不是删除元素。要完全删除数组元素，可以看一下\$pull和\$pop操作符：

```
db.readings.update({_id: 325}, {$unset: {'temp.f': 1}})
db.readings.update({_id: 326}, {$unset: {'temps.0': 1}})
```

\$rename

如果要修改key的名字，可以使用\$rename：

```
db.readings.update({_id: 324}, {$rename: {'temp': 'temperature'}})
```

也可以修改子文档的名字：

```
db.readings.update({_id: 325}, {$rename: {'temp.f': 'temp.fahrenheit'}})
```

\$setOnInsert

在upsert中，有时候要注意，不能重写某些数据。这时，若只想新增的数据就会非常有用，而不会修改数据。这就是setOnInsert操作符的由来：

```
db.products.update({slug: 'hammer'}, {
  $inc: {
    quantity: 1
  },
  $setOnInsert: {
    state: 'AVAILABLE'
  }
}, {upsert: true})
```

我们要增加某个库存项目的数量而不受状态影响，状态的默认值是'AVAILABLE'。如果执行插入，qty的值是1，状态会设置为默认值。如果执行更新，就只会插入qty。MongoDB 2.4 增加的\$setOnInsert操作符就是用于处理这种问题的。

数组更新操作符

数组在MongoDB文档模型中非常重要。自然而然地，MongoDB提供了许多操作数组的操作符。

\$push、\$pushAll 和 \$each

如果要在数组后面追加值，\$push是最好的选择。默认情况下，它会在数组尾部添加一个单独的元素。例如，在商品标签里添加一个新的标签，代码非常简单：

```
db.products.update({slug: 'shovel'}, {$push: {tags: 'tools'}})
```

如果要在一次更新里添加多个更新，则可以组合使用\$each与\$push操作符：

```
db.products.update({slug: 'shovel'},
  {$push: {tags: {$each: ['tools', 'dirt', 'garden']}}})
```

注意，可以向数组里添加任意类型的值，而不仅仅是增加同类型值。例如，7.3.2小节，可以添加一个商品到购物车数组里。

MongoDB 2.4版本之前，使用\$pushAll操作符来添加多个值到数组上。这个方法在2.4及其以后的版本里仍然可以使用，但是已经是过时的方法，因为\$pushAll可能在以后的版本中完全删除。\$pushAll操作符的使用如下所示：

```
db.products.update({slug: 'shovel'},
  {$pushAll: {'tags': ['tools', 'dirt', 'garden']}})
```

\$slice

\$slice操作符是在MongoDB 2.4里添加的，其目的是方便管理经常更新的数组。当向数组添加值但是不想数组太大的时候，这个操作符非常有用。它必须与\$push、\$each操作符一起使用，允许用来剪短数组的大小、删除旧的值。传递给\$slice的参数必须是小于或者等于0。这个参数的值是数组里允许的项目数量乘以-1。

对这个语义有点疑惑，所以来看一下具体的例子。假设要更新下面的文档：

```
{
  _id: 326,
  temps: [92, 93, 94]
}
```

则可以使用如下的命令来更新文档：

```
db.temps.update({_id: 326}, {
  $push: {
    temps: {
      $each: [95, 96],
      $slice: -4
    }
  }
})
```

这是完美的语法。这里传递了-4给\$slice操作符。在更新后，文档的数据如下：

```
{
  _id: 326,
  temps: [93, 94, 95, 96]
}
```

在推送数据到数组后，从开头删除了值，只有4个数留下。如果传递-1给\$slice操作符，结果数组是[96]。如果传递0，结果就是[]，为空数组。注意，从MongoDB 2.6开始也可以传递正整数了。如果传递给\$slice的是正整数，它就会从数组尾部开始删除元素。

在之前的例子中，如果使用了\$slice: 4，结果将是

```
temps: [92, 93, 94, 95].
```

\$sort

与\$slice很像, MongoDB 2.4新增了\$sort操作符, 帮助更新数组。当使用\$push和\$slice时, 有时候要先排序文档再删除它们。思考下面的文档:

```
{
  _id: 300,
  temps: [
    { day: 6, temp: 90 },
    { day: 5, temp: 95 }
  ]
}
```

我们有个数组子文档。当添加子文档到数组中并分割它时, 首先要确定按照日期排序, 保留更高的day值。我们可以使用如下的更新实现这个目标:

```
db.temps.update({_id: 300}, {
  $push: {
    temps: {
      $each: [
        { day: 7, temp: 92 }
      ],
      $slice: -2,
      $sort: {
        day: 1
      }
    }
  }
})
```

当运行更新时, 首先根据day排序temps数组, 这样最小的值在开头部分。然后就可以把数组分割为两个部分了。结果是保留两个更高day值的子文档:

```
{
  _id: 300,
  temps: [
    { day: 6, temp: 90 },
    { day: 7, temp: 92 }
  ]
}
```

这里使用\$sort操作符需要\$push、\$each、\$slice。虽然很有用, 但是这种处理的是非常少见的情况, 我们可能不会经常使用\$sort更新。

\$addToSet 和 \$each

使用\$addToSet也会往数组后面添加值, 但是它更加特殊: 它只会添加向数组里添加不存在的值。因此, 如果铁锹已经存在于tool标签中了, 则下面的代码就不会更新文档:

```
db.products.update({slug: 'shovel'}, {$addToSet: {'tags': 'tools'}})
```

如果需要一次添加多个值到数组里，就必须组合使用\$each 和\$addToSet操作符。这是例子代码：

```
db.products.update({slug: 'shovel'},
  {'$addToSet': {'tags': {'$each': ['tools', 'dirt', 'steel']}}})
```

只有不存在于tags的\$each值才会被追加。注意，\$each只能和\$addToSet、\$push操作符一起使用。

\$pop

从数组中删除元素的最基本的方式就使用\$pop操作符。

如果\$push追加了一个项目到数组中，\$pop会删除最后一个推进的项目。虽然它经常与\$push运算符搭配使用，但是我们可以单独使用\$pop运算符。如果数组标签tags包含值['tools', 'dirt', 'garden', 'steel']，下面的\$pop操作会删除steel标签：

```
db.products.update({slug: 'shovel'}, {'$pop': {'tags': 1}})
```

与\$unset类似，\$pop的语法是{'\$pop': {'elementToRemove': 1}}。但是与\$unset不同，\$pop接受第二个可能的值-1来删除数组的第一个元素。

下面是如何从数组中删除tools标签的代码：

```
db.products.update({slug: 'shovel'}, {'$pop': {'tags': -1}})
```

有一点可能比较遗憾的是，无法返回\$pop从数组里删除的元素。因此，\$pop的名字和实际的栈操作运算的结果不太一样。

\$bit

如果使用过按位运算的操作符，就会发现自己可能需要在更新操作里使用相同的计算。按位运算经常在单个的二进制级别来执行逻辑运算。常见的例子是（特别是C语言）使用按位运算操作来传递标志位。换句话说，如果一个整数二进制的第4位是1，就可以使用某些条件，进而执行代码。

通常有更聪明和适合的方法来解决这个问题，但是这种存储会保持最小尺寸，并满足现存系统的工作。MongoDB提供了\$bit操作符来进行按位运算，或（OR）和与（AND）在更新中都可以使用。

我们来看一下在MongoDB存储位敏感的数据，并且更新它们。UNIX文件权限通常这样存储。如果在UNIX里运行ls -l命令，就会看到drwxr-xr-x。第一个标志d表示文件是个目录，r表示文件权限，w表示写权限，x表示执行权限。这些标志有3块，分别控制不同的用户权限、用户组、每个人。因此，这个例子假设每个用户有所有的权限，但是其他只有读取和执行的权限。

有时候权限使用单个数字表示，使用二进制数据的位表示权限。x用1表示，w用2表示，r用4表示。因此你可以使用二进制格式的111或者rwx来表示7。也可以使用二进制格式的101或者使用r-x来表示5。也可以使用二进制格式的011或-wx来表示3。

我们来这些特性在MongoDB里存储一个变量。从下面这个文档开始：

```
{
  _id: 16,
  permissions: 4
}
```

4此时可以表示二进制格式的100，或者r--。也可以使用按位运算符OR来添加权限：

```
db.permissions.update({_id: 16}, {$bit: {permissions: {or: NumberInt(2)}}})
```

在JavaScript shell里，必须使用NumberInt()，因为它默认使用了double类型。结果文档保护了一个二进制格式的100和010 或OR运算，结果是110，也就是十进制的6：

```
{
  _id: 16,
  permissions: 6
}
```

对于二进制运算，我们可以使用AND代替OR。这又是一个少见的情况，不会经常使用，但是在某些特定情况下非常有用。

\$pull 和 \$pullAll

\$pull是\$pop的复杂形势。使用\$pull，可以通过值精确指定要删除的元素。回到tags的例子，如果需要删除dirt标签，不需要知道它在数组中的位置，只需要告诉\$pull操作符要删除哪个值即可：

```
db.products.update({slug: 'shovel'}, {$pull: {tags: 'dirt'}})
```

\$pullAll与\$pushAll的工作非常类似，允许提供要删除值的列表。

要同时删除dirt和garden标签，可以使用\$pullAll，代码如下：

```
db.products.update({slug: 'shovel'},
  {$pullAll: {'tags': ['dirt', 'garden']}})
```

\$pull的一个强大特性就是可以传递查询作为参数来选择要拉取的元素。思考下面的文档：

```
{_id: 326, temps: [97.6, 98.4, 100.5, 99.1, 101.2]}
```

假设要删除温度大于100的值，查询文档如下：

```
db.readings.update({_id: 326}, {$pull: {temps: {$gt: 100}}})
```

修改后的文档结构如下：


```
{_id: 326, temps: [97.6, 98.4, 99.1]}
```

定位更新

MongoDB里经常会使用子文档来建模数据，但是这些子文档操作起来并不方便，直到有了定位操作符。

定位操作符允许我们通过原点选择器来定位要更新的元素。例如，有如下的订单文档：

```
{
  _id: ObjectId("6a5b1476238d3b4dd5000048"),
  line_items: [
    {
      _id: ObjectId("4c4b1476238d3b4dd5003981"),
      sku: "9092",
      name: "Extra Large Wheelbarrow",
      quantity: 1,
      pricing: {
        retail: 5897,
        sale: 4897
      }
    },
    {
      _id: ObjectId("4c4b1476238d3b4dd5003982"),
      sku: "10027",
      name: "Rubberized Work Glove, Black",
      quantity: 2,
      pricing: {
        retail: 1499,
        sale: 1299
      }
    }
  ]
}
```

若想设置第二行项目的数量为5，使用的SKU为10027，但我们不知道line_items数组保存在哪里，也不知道它是否存在。我们可以使用简单的查询器和位置操作符来解决上面这些问题：

```
query = {
  _id: ObjectId("6a5b1476238d3b4dd5000048"),
  'line_items.sku': "10027"
}
update = {
  $set: {
    'line_items.$.quantity': 5
  }
}
db.orders.update(query, update)
```

位置操作符\$就是我们在line_items.\$.quantity字符串里看到的。如果查询选择器匹配，则SKU为10027的文档索引会内部替换位置操作符，从而更新正确的文档。

如果数据模型包含子文档，就会发现在执行细微的文档更新时位置操作符非常有用。

7.4.3 findAndModify 命令

本章介绍了许多使用findAndModify命令的例子，现在只剩下在JavaScript shell里使用这些操作参数了。这里是使用findAndModify的简单例子：

```
doc = db.orders.findAndModify({
  query: {
    user_id: ObjectId("4c4b1476238d3b4dd5000001"),
  },
  update: {
    $set: {
      state: "AUTHORIZING"
    }
  }
})
```

有许多参数可以控制此命令的功能。下面的参数是query、update、remove需要的：

- query——查询选择器，默认为{ }。
- update——指定更新的文档，默认是{ }。
- remove——布尔值，如果为true，则返回删除的对象。默认是false。
- new——布尔值，如果为true，则返回修改后的文档。默认是false，意味着返回最初的文档。
- sort——指定排序的方向。使用findAndModify一次就修改一个文档，sort参数可以帮助控制要处理哪个文档。例如，可以根据{created_at: -1}排序来处理最近创建的文档。
- fields——如果只要返回部分字段，就使用这个参数来指定它们。这在处理大文档时非常有用。这些指定可以在任意查询里指定。可以查看第5章的例子。
- upsert——布尔值，为true时，findAndModify作为upsert操作。如果文档不存在就创建它。注意，如果要返回新创建的文档，就需要指定{new: true}。

7.4.4 删除

删除文档也会有许多挑战。我们可以从集合中删除整个文档，也可以通过传递查询器给remove方法来删除部分文档。删除所有的评价文档数据则非常简单：

```
db.reviews.remove({})
```

但是通常我们可能要删除某个用户的评价：

```
db.reviews.remove({user_id: ObjectId('4c4b1476238d3b4dd5000001')})
```

所有调用`remove`方法都会使用查询选择器来指定要删除的文档。但是我们可能还会对并发和原子性有些疑问。将在下一节里解答这些问题。

7.4.5 并发、原子性和隔离

很重要的一点就是要理解MongoDB里并发的工作原理。在MongoDB 2.2之前，锁策略非常简单，单个读/写锁驻留在整个MongoDB实例中。这意味着，在任意时刻，MongoDB只允许一个写或者多个读（不是2个）操作。在MongoDB 2.2版本里改成了数据库级别的锁，意味着语义上在数据库级别而不是整个MongoDB实例级别使用锁；数据库可以有一个写者或者多个读取者。在MongoDB 3.0版本中，WiredTiger存储引擎工作在集合级别，提供了更加强大的文档级别的锁。其他的存储引擎可能提供了类似的特性。

锁特性听起来比实际更糟糕，那是因为针对锁很少有优化措施。其中之一就是数据库在内存里保留文档的内部映射。对于非RAM里的读/写，数据库都会屈服于其他操作，直到文档加入内存。

第二个优化就是针对写锁。如果某个写操作需要长时间来完成，那么所有其他的读和写操作都会被阻塞。所有的插入、更新和删除都需要写入锁。插入很少需要很长时间。但是更新会影响整个集合，还有删除也会影响许多文档，可能需要很长时间才能完成。当前的解决方案允许这种长时间运行的操作为其他读/写让路。当一个操作屈服时，它会暂停，并释放锁，后面再重新启动。

尽管对于锁机制进行了优化，但是MongoDB在大量读/写的情况下还是会受影响性能。一种好而简单的避免问题的方法就是把高并发的集合保存到单独的数据库里，特别是使用MMAPv1存储引擎时。但是正如之前提到的，这种情况在MongoDB 3.0好多了，因为WiredTiger存储引擎工作在集合级别，而不是数据库级别。

当更新和删除文档时，这种退让机制可能喜忧参半。设想我们要在其他操作发生之前更新或者删除所有的文档。此时，我们可以使用专门的参数`$isolated`来保持操作独立，不会让路。可以为查询选择器添加`$isolated`操作符：

```
db.reviews.remove({user_id: ObjectId('4c4b1476238d3b4dd5000001'),
  $isolated: true})
```

同样的代码可以应用到多个操作中。它会强制独立隔离完成多个更新。

```
db.reviews.update({$isolated: true}, {$set: {rating: 0}}, {multi: true})
```

这个更新会把每个评价设置为0。这个操作是隔离进行的，操作不会屈服于其他操作，确保系统的一致性视图。

注意，如果使用`$isolated`半途失败了，就不会有显示的回滚操作。部分文档已经更新而其他部分还是原来的值。MongoDB 2.2之前，`$isolated`操作符叫做`$atomic`，这个名字已经

过时，因为这些操作失败在于是非原子性的。`$isolated`操作符无法在分片集合中使用，意味着我们使用的时候必须注意这个问题。

7.4.6 更新性能注意事项

下面的问题只适用于MMAPv1存储引擎，它是当前默认的存储引擎。第10章讲解了WiredTiger引擎，它与MMAPv1的工作方式不同，更加高效。如果对WiredTiger感兴趣，现在就可以阅读第10章的内容！

经验表明，理解MongoDB如何更新磁盘上的文档数据可以帮助我们优化性能。首先要理解的事情就是更新发生的程度。理想情况下，更新会影响磁盘上BSON文档的最小部分，因为这样是最高效的。但是这种情况不会经常发生。

更新磁盘上的文档有三种方式。第一种更新是最高效的，当只更新文档里的单个值并且BSON文档大小不会改变时才发生。这通常发生在`$inc`操作符中。因为`$inc`只增加整数，磁盘上的文件大小不会发生变化。如果整数代表`int`，那么在磁盘上只占用4个字节，长整数和`double`类型会占用8个字节。修改这些数值不会改变存储空间的大小，只需要把文档的只重新写入磁盘即可。

第二种更新是修改文档的大小和数据结构。BSON文档表示为字节数组，前4个字节表示文档的大小。因此使用`$push`操作符修改文档，既增加文档的大小又修改结构。这需要在磁盘上重写整个文档。虽然这样还不是特别低效率，但是值得我们注意。如果有大型文档——假设4MB——我们要在此文档里添加数组，那么在服务端会是很大的工作。这意味着如果要做许多的更新操作，最好尽量保持文档较小。

第三种更新是重写一个文档。如果文档扩大，但是不能满足现在的磁盘空间，则不仅仅需要重写，还需要移动到新的空间里。这种移动操作如果经常发生，则会非常昂贵。MongoDB会通过动态调整集合分配预留空间的填充因子（padding factor）来优化这个问题。这意味着当在一个集合中有许多的需要文档迁移更新时，内部预留空间的填充因子会增加。填充因子会乘以每个插入文档的大小来获取额外的空间。这也许可以减少未来文档重新迁移的数量。此外，MongoDB 3.0使用了2的幂来作为MMAPv1存储引擎默认的记录空间分配大小。

要看下某个集合的填充因子，运行下面的命令即可：

```
db.tweets.stats()
{
  "ns" : "twitter.tweets",
  "count" : 53641,
  "size" : 85794884,
  "avgObjSize" : 1599.4273783113663,
  "storageSize" : 100375552,
  "numExtents" : 12,
  "nindexes" : 3,
```

```

"lastExtentSize" : 21368832,
"paddingFactor" : 1.2,
"flags" : 0,
"totalIndexSize" : 7946240,
"indexSizes" : {
  "_id" : 2236416,
  "user.friends_count_1" : 1564672,
  "user.screen_name_1_user.created_at_-1" : 4145152
},
"ok" : 1
}

```

这个推文tweets的集合的填充因子是1.2，这表示当插入100个字节的文档时，MongoDB会在磁盘上分配120个字节。默认的填充因子是1，表示不会分配额外的空间。

现在，简要强调下，当数据超过RAM大小或总需要极大地写入负载时，这里提到的问题特别对于部署情况要格外注意。此时，重写或者移动文档都会产生很高的成本。随着伸缩MongoDB应用，仔细思考以下最佳的更新方式，比如用\$inc来避免这些开销。

7.5 复习更新操作符

Reviewing update operators

表7.1 列举了本章之前讨论的更新操作符。

表 7.1 操作符

操作符	
\$inc	根据给定的值增加字段
\$set	设置字段为给定的值
\$unset	取消设置字段
\$rename	重命名字段为给定的值
\$setOnInsert	在 upsert 中，只在插入时设置字段
\$bit	只执行按位更新字段
数组操作符	
\$	根据查询选择器定位要更新的子文档
\$push	添加值到数组中
\$pushAll	添加数组到一个数组中。过期，被\$each取代
\$addToSet	添加值到数组中，重复了也不处理
\$pop	从数组中删除第一个或者最后一个值
\$pull	从数组中删除匹配查询条件的值
\$pullAll	从数组中删除多个值

数组运算符修饰符	
\$each	与\$push和\$addToSet一起使用来操作多个值
\$slice	与\$push和\$each一起使用来缩小更新后数组的大小
\$sort	与\$push、\$each、\$slice一起来排序数组中的子文档
隔离运算符	
\$isolated	隔离其他操作，不允许其他操作交叉更新多个文档

7.6 总结

Summary

本章我们介绍了许多内容。各种不同的更新方式，这些操作展现出的强大功能都是可以保证的。事实上，MongoDB更新语言与查询语言一样复杂。我们可以像更新简单文档一样更新复杂的文档。当需要的时候，也可以单独原子地更新某个文档，与findAndModify一起使用，可以构建事务性工作流。

如果你已经学习完本章，感觉自己可以独立应用这些例子了，那么现在你就在成为MongoDB大师的路上了。

第三部分 精通 MongoDB

MongoDB mastery

已经阅读了本书的前面2个部分，从开发者角度我们应该对MongoDB有了很好的理解。现在要更上一层楼了。在本书的最后部分里，我们会从数据库专家的角度来看待MongoDB。这意味着我们需要了解数据库的性能、部署、容错和伸缩性等所有的知识。

要获得MongoDB的最佳性能，就必须设计出高效的查询并确保正确、恰当地索引。这是我们将要在第8章里学习的内容。我们将会看到为什么索引是如此重要，并学习如何选择索引，查询优化器时如何使用索引。我们还会学习如何使用一些帮助工具，比如查询解释器和监控器。我们将会发现创建和优化的查询一定与文本搜索有关系。要简化并强化这个功能，MongoDB提供了文本搜索专用的特性，我们会在第9章里介绍。这些特性允许我们为单词和短语编写智能查询。

第10章是关于WiredTiger存储引擎的底层原理。第11章深入介绍了复制功能。我们会花费大量篇幅来介绍可复制集群的工作原理，以及如何部署高可用和自动故障转移的集群。此外，我们还会学习如何使用复制来伸缩应用程序的读能力，以及自定义写持久化。

水平扩展是现代数据库系统的利器。MongoDB通过分片方式实现了水平分区数据。第12章会深入介绍数据库分片的理论和实战，以及如何设计schema和如何部署分片集群数据库。

最后一章介绍了部署和管理知识。在这一章里，我们会推荐专门的硬件和操作系统，会深入介绍如何备份、监控和分析调试 MongoDB 集群。

然后就是索引的实战了。我们将会讨论唯一的、轻便的以及多键索引，并且提供索引管理的一些知识。接下来，会深入介绍查询优化器是如何使用索引的以及如何使用索引来优化查询。

在3.1.2节中，我们介绍了MongoDB加入全文索引的官方插件，能支持在索引文档中匹配某个字段的值或者某个短语。举个例子，你也许会想将一天中所有单词的全文索引。这就需要全文索引。这将在第9章介绍。而在此之前，使用空间地理位置索引来查找基于位置的文档（经纬度和高度）。本章会介绍索引的核心概念，帮助大家对索引的创建开发与维护。这样大家不仅会创建索引，还会懂得如何高效地优化查询。

8.1 索引理论

Indexing theory

我们会继续前进，深入探讨一个数据库的索引如何工作。索引是数据库索引的基石。

索引与查询优化

Indexing and query optimization

本章内容

- 基本索引概念和理论
- 索引管理实践建议
- 使用组合索引进行复杂查询
- 优化查询
- 所有的MongoDB索引参数

索引非常重要。正确设置了索引，MongoDB可以高效地使用它的硬件设备，并且快速服务查询。但是错误的索引带来相反的结果：查询减速、写减速、恶化硬件设备的使用。所以，对于任何想要高效使用MongoDB的人来说都必须理解索引机制。

对于许多开发者来说，索引是非常神秘的。其实并非如此。一旦你学习完本章内容，就会清晰地掌握索引的知识。要介绍索引的概念，我们将会从一个实验例子开始，然后深入介绍一些索引的核心概念，并且介绍MongoDB索引背后的底层数据结构B-tree。

然后就是索引的实战了。我们将会讨论唯一的、松散的以及多键索引，并且提供索引管理的一些知识。接下来，会深入介绍查询优化，包括如何使用`explain()`以及如何使用查询优化器。

在2.0、2.4、2.6、3.0版本中，MongoDB加入了更强大的索引机制。绝大部分查询值需要索引匹配某个字段的值或者某个范围的值。但是，你也许想运行一个关于某个单词的全文查找。这就需要全文索引，这将在第9章介绍。或者我们会使用空间地理位置索引来查找某个点附近的文档（经度和纬度）。本章会介绍索引的核心概念，帮助大家进行索引的高级开发与优化，这样大家不仅会创建索引，还会懂得如何高效地优化查询。

8.1 索引理论

Indexing theory

我们会循序渐进、深入浅出，从一个扩展的类比的例子开始，到MongoDB关键实现细节结束。

在这个过程中，我们将会定义并提供许多与概念相应的例子。如果你不熟悉复合索引、虚拟内存、索引数据结构，那么本节非常有用。

8.1.1 精心策划的实验

要理解索引，则大脑中需要有个画面。想象有一本菜谱，它不是普通的做菜的书籍，而是一本非常厚的书：5000页，包含了各种不同场合下，采用各种烹饪法、配料，以及不同季节食用的菜谱宝典。这本菜谱大全我们就叫神厨宝典（Cookbook Omega）。

虽然这本书可能是最好的烹饪书籍，但是作为神厨宝典（Cookbook Omega）还有两个问题。首先是食谱是随机排序的，如在第3475页有澳大利亚炖鸭的做法，在第2页有萨卡特卡斯玉米饼的做法。

这个问题是可管理的，但不像第二个问题：《神厨宝典》（Cookbook Omega）没有索引。

这是第一个要问自己的问题：没有索引，如何在书里找到Rosemary Potatoes的做法？唯一的做法就翻遍全书直到找到这个菜谱为止。如果这个菜谱在第3973页，就要翻过许多页面。最坏的情况下，菜谱在最后一页，我们必须翻遍整本书！

这太疯狂了。其实解决方案就是创建一个索引。

简单的索引

有几种可以搜索食谱的方式，其中按名称是个不错的方法。如果给每个食谱名字的字母顺序排序，创建索引，我们就可以有一个基于名称的索引。部分菜谱的索引入口如下所示：

- Tibetan Yak Soufflé: 45
- Toasted Sesame Dumplings: 4,011
- Turkey à la King: 943

只要知道菜谱的名字（即使是首字母），也可以使用索引快速查找到菜谱。如果这是唯一期望的搜索菜谱的方式，那么现在的工作已经完成了。

但是这不太现实，我们也可以想像基于厨房的配料来查找菜谱，或者根据烹饪法来查找菜谱。对于这种情况，就需要更多的索引。

还有第二个问题：只有一个菜谱名字的索引，如何找到所有的与菜花相关的菜谱？还是缺少适当的索引！我们必须再次翻遍所有的5000页图书。对于使用配料或者食材来查询也是一样的问题。

我们需要建立其他索引，这次是配料。这个索引里使用配料名的字母顺序，每个索引入口记录配料所在的页面号码。最基本的配料索引部分内容如下所示：

- Cashews（腰果）：3; 20; 42; 88; 103; 1,215;...
- Cauliflower（菜花）：2; 47; 88; 89; 90; 275;...
- Currants（黑醋栗）：1,001; 1,050; 2,000; 2,133;...

这是不是你要找的索引？是不是很方便？

复合索引

这个索引对于要根据食材来查找所有菜谱列表非常有用。但是如果要在查询中包含菜谱的其他信息，就仍然需要再次扫描——一旦知道菜花引用的页面，就需要跳转到具体的页面来查找菜谱的名字和做法。这比翻遍整本书好得多，但是还可以做得更好。

例如，我们在《神厨宝典》里随机发现了一个很棒的菜花食谱，但是几个月后忘记了它的名字，看到它肯定会认出来。现在有两个索引，一个是基于菜谱名字的，一个是基于配料的。你能想像出来组合使用这两种索引来查找失传已久的菜花菜谱的方法吗？

事实上，不太可能。如果从菜谱名字开始，但是不记得菜谱名字，那么基于这个索引搜索只比翻遍全书好一点点而已。如果从配料索引开始，则会有一个对应的列表要去检查，而这些页码可以任意顺序插入菜谱名字的索引里。因此，我们只能使用这个索引，而且配料的索引更加有帮助。

每个查询一个索引

用户通常认为2个字段的查询可以使用2个不同字段索引来实现。存在这样的算法：查找索引中匹配关键字的页面，然后通过扫描这些页面来查找具体的菜谱信息。许多页面不符合要求，但是我们可以缩小页面扫描的范围。MongoDB 2.6之后开始支持索引交集。注意，无论使用组合索引还是索引交集，哪个更高效取决于特定的查询和系统。还有，如果可以，数据库为单个查询使用单个索引，如果经常查询多个字段，就要确保在这些字段上存在复合索引。

你能做什么？高兴的是，你可以找到失传已久的菜花菜谱，使用的方法是复合索引。

目前创建的两个索引都是单件索引，都是基于一个键值对菜谱排序。我们选择为《神厨宝典》建立一个新的索引，但是这次我们会使用两个键值而不是一个。类似这种多个键值的索引叫做复合索引。

复合索引使用了配料和菜谱名字。我们会注意到索引的名字为：ingredient-name。

索引的部分如图8.1所示。

索引值对于人类来说显而易见。我们可以根据配料搜索，也可能找到需要的菜谱，虽然可能我们只记得部分名字。

对于机器来说，这种用例也有价值，避免数据库扫描每个菜谱名字。复合索引特别有用，尤其是类似《神厨宝典》包含几百（或者几千）个菜花菜谱的时候。你能理解为什么吗？

提示一下：使用复合索引时顺序非常重要。设想一下反向的组合索引name-ingredient，这些索引是不是可以互换？

Cashews
Cashew Marinade
1,215
Chicken with Cashews
88
Rosemary-Roasted Cashews
103
Cauliflower
Bacon Cauliflower Salad
875
Lemon-baked Cauliflower
89
Spicy Cauliflower Cheese Soup
47
Currants
Creamed Scones with Currants
2,000
Fettuccini with Glazed Duck
2,133
Saffron Rice with Currants
1,050

图 8.1 菜谱书籍里的复合索引

当然不能互换。使用新的索引，一旦有了菜谱名字，搜索就限制在单个菜谱上了；书籍中的单个页面。如果使用索引搜索菜谱Cashew Marinade 和配料Bananas（香蕉），索引会告诉你没有这个菜谱存在。但是这个用例恰恰相反：我们知道配料，但是不知道菜谱名字。

现在《神厨宝典》有3个索引：一个是菜谱recipe的名字，一个是配料ingredient，一个是组合索引ingredient-name。这意味着我们可以在ingredient上删除一个索引。为什么？因为针对单个配料的查询可以使用索引ingredient-name。如果知道配料，就可以利用这个组合索引获取所有包含配料的页码列表。再看一下这个索引的例子入口就明白为什么了。

索引规则

本节的目标是给大家一个扩展的类比例子，帮助大家更好地理解索引机制。从这个比喻，可以派生出一些新的概念：

- (1) 索引可以大大减少要处理的文档数量。没有适当的索引，唯一满足条件的查询方式就是扫描全部文档，直到找到满足条件的查询。这通常指的是查询整个集合。
- (2) 唯一的单键索引将会用来处理查询^[1]。对于包含多个键的查询（配料ingredient和菜谱名recipe name），包含这些键的复合索引是最好的解决方案。
- (3) 如果ingredient配料上有索引，并且还有一个复合索引ingredient-name，就可以考虑删除一个索引。更具体地说，如果有一个复合索引a-b，那么a上的索引就是多余的，b上的不多余。
- (4) 复合索引的键值顺序很重要。记住，菜谱宝典的比喻例子到此结束了。它就是为了解索引而建模的例子，但是并非完全展现了MongoDB索引的全部特性。在下一节里，我们将会详细介绍MongoDB索引机制和建立索引的规则。

8.1.2 核心索引概念

前面的例子引出许多核心的索引概念。本章的其余部分，我们将会解释这些技术概念。

单键索引

使用单键索引，每个索引入口对应文档索引里的单个值。默认的索引在_id字段上，这是最好的单键索引的例子。因为这个字段建立了索引，每个文档的_id也存在于索引中，所以可以加速根据此字段的查询。

复合索引

虽然从MongoDB 2.6开始，可以在一个查询里使用多个索引，但是最好还是只使用一个索引。不过，如果经常需要查询多个字段，且又希望改进性能，例如，已经在电商例子的商品集合上建立了2个索引：manufacturer制造商和price价格，那么我们可以创建完全不同的数据结构。遍历时的顺序如图8.2所示。

现在，设想象一下我们有这样的查询：

```
db.products.find({
  'details.manufacturer': 'Acme',
  'pricing.sale': {
    $lt: 7500
```

^[1]一个例外是使用\$or操作符查询。但是作为通用规则，在MongoDB中这是不可能的，甚至是不可取的。

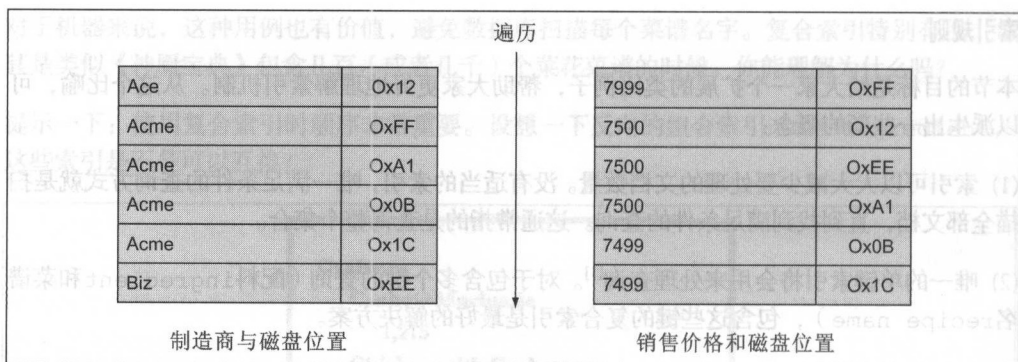


图 8.2 单键索引遍历

```

    }
  })

```

这个查询要找出Acme公司制造的价格低于\$75.00美元的所有商品记录。如果在manufacturer和price中使用其中一个单键索引查询、操作，那么查询优化器会找到其中一个更高效的，但是每个都会给我们理想的结果。使用索引来满足这些查询，需要单独遍历每个数据结构，找到它们的磁盘位置，计算交集。

复合索引是每个入口由多个键值组成的单个索引。如果要在manufacturer（制造商）和price（价格）上构建复合索引，结果会如图8.3所示。

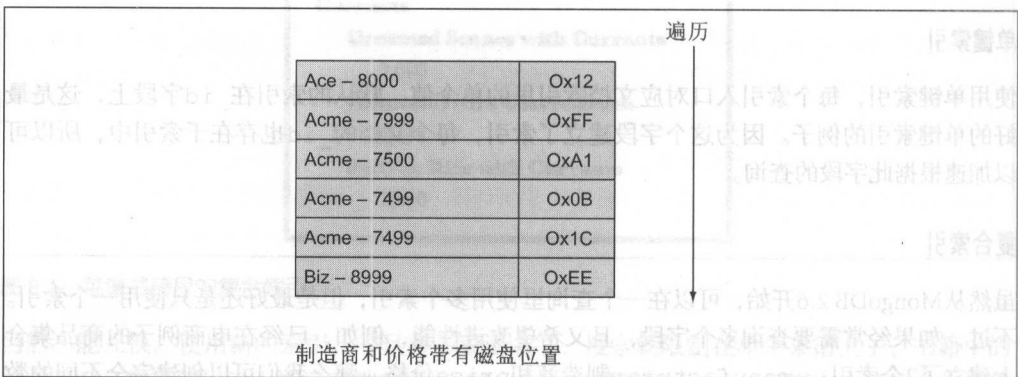


图 8.3 复合索引遍历

要满足查询要求，查询优化器需要找到索引中制造商manufacturer是Acme并且价格是7500的第一个入口。从那里开始，结果可以使用连续的扫描查找出来。

我们应该注意索引和查询一起工作的两件事情。第一，索引键值的顺序非常重要。如果我们定义的复合索引的第一个值是price，第二个是manufacturer，那么查询效率就会非常低。为什么？来看一下图8.4所示的索引入口结构。

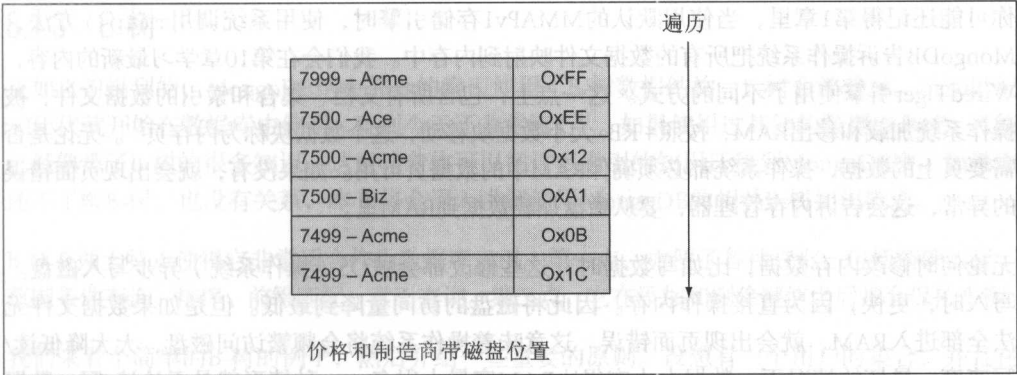


图 8.4 复合索引键值顺序调整

键值必须按照出现的顺序比较。不幸的是，这个索引没有提供索引Acme公司产品的快捷方式，所以唯一的满足查询要求的方式就是查找所有价格低于\$75.00的商品，然后选择由Acme公司制造的商品。为了弄清楚这个问题，假设你的集合有100万件商品，所有的价格低于\$100.00美元，并且按价格均匀分布。在这些情况下，查询需要扫描750000个索引入口。相比之下，使用最初的组合索引，manufacturer跟着price，入口扫描的数量和返回的入口数量是一样的。这是因为一旦到达(Acme-7500)入口，它很简单，通过顺序地扫描来进行查询。

复合索引的顺序非常重要。如果清楚了这一点，就应该明白第二件事情，即为什么我们选择的是第一个索引而不是第二个索引。图示已经非常明显，但是还有另一个看待此问题的方式。再看一下查询：两个查询词语指定不同的匹配。在manufacturer（制造商）上，我们要精确匹配关键字；在price（价格）上，我们要查询某个范围的值，从7500开始。按通常的规则，查询需要词语的精确匹配，第二个指定组合索引键值的范围。我们会在查询优化一节里复习这个概念。

索引效率

虽然索引对于查询性能非常重要，但每个新的索引都需要额外的维护成本。无论何时，集合里添加新的文档，每个集合的索引就必须修改以包含新的文档。如果某个索引包含10个索引，那么除了编写文档，每次插入都需要单独修改10个数据结构。这适用于任意写操作，无论是删除文档，还是因为空间不足挪动文档或者更新文档的索引键。对于读取密集型的应用，索引的成本是可以理解的。知道索引也有成本，所以必须仔细选择。这意味着确保所有的索引都会被使用，不会有冗余。我们可以通过监控应用系统的查询来调整索引。这会在本章的后面介绍。

还有第二个要考虑的问题。即使所有的索引都建设得恰当，也可能无法加快查询。这会在索引和数据集没有加载到RAM的时候发生。

MMAPVI 存储引擎索引使用了 mmap，索引作为文件的子集存储。索引和文档存储在相同的文件系统中，索引和文档的存储是紧密耦合的，这确保了索引和文档的加载和卸载是同步的。

你可能还记得第1章里，当使用默认的MMAPv1存储引擎时，使用系统调用mmap()方法，MongoDB告诉操作系统把所有的数据文件映射到内存中。我们会在第10章学习最新的内容，WiredTiger引擎使用了不同的方式。这一点上，包含所有文档、集合和索引的数据文件，被操作系统加载和移出RAM，按照4 KB 大小数据块移动，这个数据块称为内存页^[1]。无论是否需要页上的数据，操作系统都必须确保RAM中的数据页可用。如果没有，就会出现页面错误的异常，这会告诉内存管理器，要从磁盘加载数据到RAM里。

无论何时修改内存数据，比如写数据时，这些修改都会被OS（操作系统）异步写入磁盘。写入时，更快，因为直接操作内存。因此将磁盘的访问量降到最低。但是如果数据文件无法全部进入RAM，就会出现页面错误。这意味着操作系统将会频繁访问磁盘，大大降低读/写速度。最坏的情况下，数据大小变得比RAM容量大很多。一种情形就是无论读/写，数据都必须从磁盘或者向磁盘写入数据。这种情况有个专业的称呼“颠簸”，它会严重导致性能变慢。

幸运的是，这种情形相对容易避免。至少我们可以确保索引加入RAM里。这也是为什么不创建不需要的索引的原因。设置的索引越多，就需要越多的RAM来维护这些索引。沿着相同的路线，每个索引应该只包含需要的键值。有时候也会需要3个键值的复合索引，但是要知道它比单键索引需要更多的空间。创建一个或者2个字段索引，是为频繁地查询创建覆盖索引（covering index）。覆盖索引是所有的查询都可以使用一个查询来满足查询的索引，让查询变得非常快。覆盖索引会在8.3节里深入讨论。

记住，索引单独存储在RAM里，而且无法聚集。在聚集索引里，索引的顺序与基础数据的存储顺序一一对应。如果聚集索引使用菜谱的名字，那么所有A开头的菜谱存储在一起，后面跟随的是以B、C开头的菜谱信息，以此类推。这不符合MongoDB的实际情况。菜谱索引里的名字可以在索引里重复，名字的顺序和数据的顺序没有一一对应关系。当扫描一个使用了索引存储的集合数据时这一点非常重要，因为这意味着每个查询的文档可以在集合的任意位置上。不能保证在之前下载数据的地方。

理想情况下，索引和操作的数据集都加载到内存里，但是预估需要的内存有点困难。我们可以使用stats命令来查询索引的大小。操作的数据集是总数据的子集，每个应用程序都不一样。假设有100万用户。如果只有一半活跃用户（因此需要查询一半的用户），那么要操作的数据集合是总大小的一半。如果这些文档均匀地分布在整个集合中，那么很多不需要的文档也会加载到内存中，这需要大量额外的开销。

在第10章中，我们将会复习工作集的概念，并且会看一下专门诊断硬件性能的问题。现在，知道了添加额外索引需要的成本，就会注意加载到RAM内存中的索引和操作的数据集合的大小。这样做可以帮助我们在数据增长的情况下维护数据库良好的性能。

^[1] 4KB 大小的页是标准的，但不是通用标准。

8.1.3 B-树

正如之前提到的，MongoDB绝大部分的索引使用了B-树数据结构。B-树非常普遍，从20世纪70年代就开始在数据库中使用，直到今天还非常流行^[1]。如果使用过其他的数据库系统，可能已经熟悉了B-树的很多知识。这意味着你可以把许多索引的知识迁移到MongoDB中。如果你还不了解B-树，也没有关系，本节将会深入讲解使用MongoDB的相关B-树知识概念。

B-树有两大特点使得它非常适合作为数据库索引。第一点，方便了各种查询，包括精确匹配、范围条件查询、排序、前缀匹配、索引查询。第二点，它在添加和删除键值之后都会保持平衡。

我们来看个简单的B-树的例子，然后介绍一些重要的原则。设想有一个用户的集合，并且在名字和年龄上创建了复合索引。B-树的抽象表示如图8.5所示。

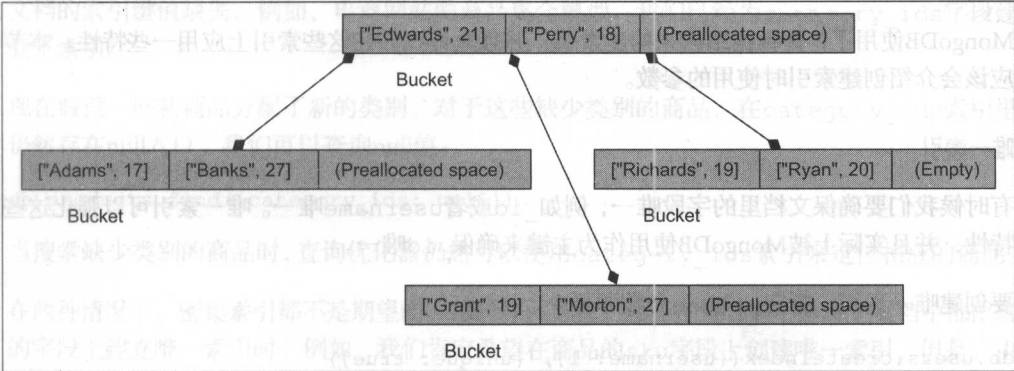


图 8.5 B-树的例子

B-树，正如你猜测的，是个树形的数据结构。树上的每个节点都可以包含多个键值。我们看到树根部分包含2个键值，每个键值都使用BSON对象表示users集合中的索引键值。在读取根节点的内容时，我们可以看到两个文档的键值，表示姓Edwards 和 Perry，还有各自的年龄21 和18。每个键值包含2个指针：一个是数据文件，一个是子节点。此外，这些节点本书也指向了包含更小值的节点。

在MongoDB的B-树实现中，新的节点分配8 192字节，这意味着，每个节点可能包含几百个键值。这个数量取决于平均的索引键值大小。此时，平均键值大小可能是30个字节。最大的键值在MongoDB 2.0以后限制到1024字节。每个键值还需要18个字节的额外开销，每个节点需要额外40个字节的开销，这样每个节点大约有170个键值。值得注意的一件事是每个节点都有一些空间（不是为了伸缩）。这些数字是相关的，这样用户就知道为什么索引大小是这样的。现在我们知道每个节点是8KB，就可以预估每个节点的键值数量。要计算这个数量，还要记住，B-树的节点通常会保持60%的饱和度。

^[1] MMAPv1 存储引擎索引使用了 B-树；集合作为双向链表存储。正如第 10 章看到的，WiredTiger 存储引擎不太一样，虽然它也使用了 B-树，但是 MMAPv1 仍然是默认的 MongoDB 存储引擎。

基于这些信息，现在应该知道了为什么索引是有成本的，需要空间和时间来更新它们。基于这些知识，我们可以在集合上创建索引，并知道什么时候避免使用索引。

8.2 索引实战

Indexing in practice

学习了这些理论之后，现在就可以来实际强化MongoDB的索引概念了。然后我们再进行一些索引管理的详细操作。

8.2.1 索引类型

MongoDB使用了B-树数据结构来构建索引，并且允许我们在这些索引上应用一些特性。本节应该会介绍创建索引时使用的参数。

唯一索引

有时候我们要确保文档里的字段唯一，例如 `_id` 或者 `username` 唯一。唯一索引可以强化这些特性，并且实际上被MongoDB使用作为主键来确保 `_id` 唯一。

要创建唯一索引，指定 `unique` 参数即可：

```
db.users.createIndex({username: 1}, {unique: true})
```

唯一索引会在所有的文档中确保唯一性。如果使用已经存在的 `username` 值插入 `users` 集合中，会抛出下面的异常：

```
E11000 duplicate key error index:
  gardening.users.$username_1 dup key: { : "kbanker" }
```

如果使用驱动，这个异常只有使用安全驱动模式插入数据时才会被捕获，这是默认的设置。在尝试插入两个相同 `_id` 的文档时也会遇到这个错误——每个MongoDB集合在此字段上都有唯一的索引，因为它是主键。

如果在集合上需要唯一索引，最好在插入数据之前创建索引。提前创建索引，可以确保从一开始就建立约束。当在包含数据的集合上创建唯一索引时，可能会出现失败，因为在集合中可能存在重复的数据。当重复的键值存在时，索引创建失败。

如果发现需要在已有的集合上创建唯一索引，可以有多个选择。首先可以重复尝试创建唯一索引，并且使用错误信息来删除重复的键值。如果数据不是很重要，也可以使用 `dropDups` 参数来告诉MongoDB删除重复的键值。例如，如果 `users` 集合已经包含数据，而且不关注删除的重复键值文档，就可以使用如下的命令来完成操作：

```
db.users.createIndex({username: 1}, {unique: true, dropDups: true})
```


注意使用 dropDups

注意，要保存的重复键的文档是随机的，所以使用这个功能时要格外小心。通常情况下，我们要决定而不是MongoDB决定删除哪个重复的文档。

从MongoDB 3.x开始删除了dropDups参数，而且还没有替代品。可以创建新的集合，在新的集合上创建唯一索引，从旧的集合中复制所有的文档到新的集合中（确保在复制过程中忽略错误）或者手动处理重复的键值。

稀疏索引

索引默认是密集型的。这意味着，对于索引集中的每个文档，存在一个对应的入口，即使文档的索引键值缺失。例如，电商网站的商品集合模型，我们已经为category_ids字段建立了索引。

现在假设一些新商品分配了新的类别。对于这些缺少类别的商品，在category_ids索引里仍然存在null入口。我们可以查询null值：

```
db.products.find({category_ids: null})
```

当搜索缺少类别的商品时，查询优化器仍然可以使用category_ids索引来定位相应的商品。

在两种情况下，密集索引都不是期望的。第一种，当我们想要在集合里不是每个文档中都出现的字段上建立唯一索引时。例如，我们肯定希望在商品的sku字段上创建唯一索引，但是，由于某些原因，商品进入集合后，sku还没有赋值。如果有唯一索引而尝试插入两个sku为null的商品文档，则第一个会成功，后续的所有插入都会失败，因为sku为null的索引入口已经被占用了。这种情况下，不适用于密集索引。相反，我们需要一个唯一的稀疏索引(sku为null)。

在稀疏索引中，只有包含某些索引键值的文档才会出现。如果要创建稀疏索引，要做到指定{sparse:true}。例如，可以在sku上创建稀疏索引，如下所示：

```
db.products.createIndex({sku: 1}, {unique: true, sparse: true})
```

还有一种使用稀疏索引的情况：集合中大量的文档不包含所有键值时。例如，假设允许匿名用户浏览评价电商网站时，可能一半的用户评价是没有user_id字段的，但是如果对它做索引，就有一半的入口是null。这样比较低效。原因有两个，首先会增加所有的大小；其次会在添加或者删除null user_id的文档时需要更新索引。

如果很少或者从不查询匿名评价，也许可以在user_id上建立稀疏索引。而且，设置sparse参数非常简单：

```
db.reviews.createIndex({user_id: 1}, {sparse: true, unique: false})
```

现在，只有包含user_id外键的评价会创建索引。

多键索引

之前的章节里，我们看了几个索引字段为数组的例子^[1]。这个可以通过多键索引实现，它允许在索引里使用多个入口来连接同一个文档。如果我们看一个简单的例子就会发现它有意义。假设我们有一些标签的商品文档，如下所示：

```
{
  name: "Wheelbarrow",
  tags: ["tools", "gardening", "soil"]
}
```

如果在tags上创建索引，每个文档tags数组的值会出现在索引里，就意味着针对这些数组任意值在索引上的查询都会定位到文档上。这是多键索引背后的思想：多个索引入口或者键值，引用同一个文档。

MongoDB通常采用多键索引，也有一些异常，比如，对于哈希索引。无论什么时候对数组建立索引，每个数组值都会作为文档的入口。

智能使用多键索引对于MongoDB schema设计至关重要。这可以在第4章到第6章的例子中得到证实；附录B里提供了更多的例子。不过创建、更新或者删除多键索引比单件索引更加昂贵。

哈希索引

在之前B-树索引的例子中，我们展示了MongoDB如何创建索引树。因此，在菜谱的索引里，“Apple Pie”的索引入口靠近“Artichoke Ravioli”入口。这也许显而易见，而且非常自然。但是MongoDB也支持哈希索引，这里的入口首先通过哈希函数来确定^[2]。这意味着哈希值决定顺序，所以这些菜单在索引里不会彼此靠近。

这种索引可以在MongoDB里通过传递‘hashed’参数来创建。例如：

```
db.recipes.createIndex({recipe_name: 'hashed'})
```

因为索引值是最初参数的哈希值，所以这些索引也有一些限制：

- 等值查询相似，不支持范围查询。
- 不支持多键哈希。
- 浮点数在哈希之前转换为整数，因此，4.2和4.3有相同的哈希索引。

知道了这些限制，你可能好奇：为什么这么多人使用哈希索引？答案是，哈希索引上的入口是均匀分布的。换句话说，当有键值数据不均匀分布时，哈希函数可以创建均匀性。“Apple Pie”和“Artichoke Ravioli”在哈希索引中就不会相邻了。索引数据的位置已经变化了。它对于分片集合非常有用，分片索引决定文档分配到哪个片中。如果分片索引基于增长的值，比如

^[1]例如考虑一下类别ID。

^[2]回忆下哈希函数，接受输入参数并映射到固定长度的区域中。对于给定的输入值，输入结果是连续的。好的哈希函数会均匀分布输出值，这样看起来比较随机。

MongoDB OIDs^[1]，那么新创建的文档只会插入单个片中，除非索引是哈希的。

我们来深入研究这个问题。除非显示设置，否则MongoDB文档会使用OID作为主键。这就是一组连续生成的OID：

```
5247ae72defd45a1daba9da9
```

```
5247ae73defd45a1daba9daa
```

```
5247ae73defd45a1daba9dab
```

注意，这些值多么相似！这是因为最重要的位是基于创建时间生成的。当新的文档使用这些ID插入时，它们的索引入口会彼此相近。如果使用这些ID来决定文档保存到哪个片（机器）中，那么这些文档很可能在同一个机器上。这样是非常有害的，当集合接受大量的写请求时会产生大量的负载压力，因为只有一台机器处理这些请求。

哈希索引通过均匀分散这些文档来解决这个问题，因此可以跨片或者跨机器存储。要完全弄明白这个问题，等到阅读第12章就可以了。

如果对于分片存储（sharding）不熟悉，在第11章有更详细的介绍。之后就更加容易理解这个例子了。

现在，重要的事情是记住哈希索引改变了索引入口的位置，这对于分片集合非常有用。

地理空间索引

另外一个有用的功能就是查询某个地点附近的文档，基于经纬度来存储每个文档。例如，用户非常希望查找家附近的餐馆时，一个办法就是查询所有10英里半径内的索引餐馆。执行这个查询就需要索引可以高效地计算出地理位置的距离，包括地球的弧度。地理位置索引可以处理这种问题。

8.2.2 索引管理

在本节和前面的章节里，我们已经讨论了简单的索引管理，比如创建索引。当在真实项目中使用索引时，深入理解本节的主题非常重要。这里我们来看一下创建和删除索引，以及如何处理压缩和备份。

创建及删除索引

到目前为止，我们已经创建了一些索引，感觉这个比较简单，在shell里简单地调用`createIndex()`方法即可，或者使用驱动接口。请注意，在MongoDB 3.0里，`createIndex()`已经取代了之前使用的`ensureIndex()`方法。你可能不知道的是，这个方法的工作原理是创建一个带有新索引的文档并把它存储到专门的`system.indexes`集合里。

^[1]MongoDB 对象 ID（OID）是 MongoDB 默认使用的 ID。我们在第 3 章 3.2.1 节有详细的介绍。

通常使用帮助方法创建索引更加简单，但是也可以手动创建索引（这也是帮助方法的工作）。我们需要确保键值的最小集合：`ns`，`key`，`name`。`ns`是命名空间，`key`是字段或者字段的组合，`name`是引用索引的名字。其他的参数，例如`sparse`，也可以指定。

例如，我们在`users`集合上创建稀疏索引：

```
use green
spec = {ns: "green.users", key: {'addresses.zip': 1}, name: 'zip'}
db.system.indexes.insert(spec, true)
```

如果插入没有返回错误，索引现在就存在了，我们可以通过查询`system.indexes`集合来证实这一点：

```
db.system.indexes.find().pretty()
{
  "ns" : "green.users",
  "key" : {
    "addresses.zip" : 1
  },
  "name" : "zip",
  "v" : 1
}
```

MongoDB 2.0添加了`v`字段来保存索引的版本。这个版本字段允许内部索引格式将来的修改，但是开发者其实不太需要关注这些。

要删除索引，也许会认为只要从`system.indexes`集合里删除索引文档就可以了。其实这个操作是被禁止的。必须使用`deleteIndexes`命令来删除数据库。正如索引的创建一样，这里有删除索引的帮助类。如果你要运行命令本身，也可以这样做。命令需要一个包含集合名字或者索引名字的文档作为参数来删除索引，“*”用来删除所有索引。要手动删除创建的索引，可以使用如下命令：

```
use green
db.runCommand({deleteIndexes: "users", index: "zip"})
```

绝大部分情况下，我们将会使用shell帮助方法来创建和删除索引：

```
use green
db.users.createIndex({zip: 1})
```

我们也可以使用`getIndexSpecs()`方法来检查索引规范：

```
> db.users.getIndexSpecs()
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "ns" : "green.users",
    "name" : "_id_"
  }
]
```

```

    },
    {
      "v" : 1,
      "key" : {
        "zip" : 1
      },
      "ns" : "green.users",
      "name" : "zip_1"
    }
  ]
}

```

最后,我们也可以使用dropIndex()方法来删除索引。注意,必须提供索引的名字参数:

```

use green
db.users.dropIndex("zip_1")

```

创建索引时也可以提供自己的名字参数。

这是创建和删除索引的基础知识。对于创建索引时期望的内容,需继续阅读。

构建索引

绝大部分时间,我们都想在从部署到生产环境之前定义索引。这样允许随着数据的插入增量式构建索引。但是,也有两种情况,我们可能会选择事后构建索引。第一种是当我们还没有发布到生产环境时导入大量数据。例如,必须把应用程序迁移到MongoDB并需要从数据仓库导入用户信息进来时,我们需要事先为用户数据创建索引,这样做可以从开始确保索引处于理想状态下的平衡度和压缩度。

第二种情况(更加显而易见)是当我们必须优化新查询时,在现有数据集上创建索引。当我们添加或者修改功能时会发生这个事情,而且是比我们想像的次数还多。假设我们允许用户使用username登录,所以要对username字段建立索引。然后我们又修改程序允许用户使用email来登陆;现在我们需要在email上简历第二个索引。注意这种情况,因为这样做需要重新考虑索引。

不管是为什么创建新索引,过程通常都不愉快。对于大型数据集,构建索引需要几个小时,甚至几天。当然我们可以通过MongoDB日志来监控构建索引的过程。我们来看一下数据集构建索引的例子(我们会在下一节里使用)。首先,我们声明要构建的索引:

```

db.values.createIndex({open: 1, close: 1})

```

注意生命索引

因为声明索引非常简单,所以促发索引构建也非常简单。如果数据集足够大,构建过程会花费很长时间。在生产环境下,这可能是个噩梦,因为没有办法中止这个过程。如果确实发生了,可能要筹备到从服务器上。最谨慎的忠告是把索引构建当做数据库迁移一样对待,要格外小心。

索引构建分为两步。第一步，对构建索引的值进行排序。排序过的值插入B-树里的效率更高。注意，排序的进度使用有序文档占全部文档的比例表示：

```
[conn1] building new index on { open: 1.0, close: 1.0 } for stocks.values
1000000/4308303 23%
2000000/4308303 46%
3000000/4308303 69%
4000000/4308303 92%
Tue Jan 4 09:59:13 [conn1] external sort used : 5 files in 55 secs
```

第二步，排序后的值插入索引中。进度显示方式相同，构建索引花费的时间使用插入system.indexes的时间来指示：

```
1200300/4308303 27%
2227900/4308303 51%
2837100/4308303 65%
3278100/4308303 76%
3783300/4308303 87%
4075500/4308303 94%
Tue Jan 4 10:00:16 [conn1] done building bottom layer, going to commit
Tue Jan 4 10:00:16 [conn1] done for 4308303 records 118.942secs
Tue Jan 4 10:00:16 [conn1] insert stocks.system.indexes 118942ms
```

除了检查MongoDB日志，也可以使用shell的currentOp()方法来检查索引构建的进度。这个命令输出各种不同的信息，看起来如列表8.1所示^[1]。

列表8.1 使用shell currentOp()方法来检查索引构建的过程

```
> db.currentOp()
{
  "inprog" : [
    {
      "opid" : 83695,
      "active" : true,
      "secs_running" : 55,
      "op" : "insert",
      "ns" : "stocks.system.indexes",
      "insert" : {
        "v" : 1,
        "key" : {
          "desc" : 1
        },
        "ns" : "stocks.values",
        "name" : "desc_1"
      },
      "client" : "127.0.0.1:56391",
      "desc" : "conn12",
      "threadId" : "0x10f20c000",
      "connectionId" : 12,
      "locks" : {
        "^" : "w",
        "^stocks" : "W"
      }
    }
  ]
}
```

操作 ID

显示这是在查询中使用索引

股票数据库

与操作关联的锁

^[1]注意，如果你已经开始从 MongoDB shell 构建索引，就必须打开新的 shell 窗口来运行 currentOp 命令。db.currentOp()的更多信息请参考第 10 章。

```

    },
    "waitingForLock" : false,
    "msg" : "index: (1/3) external sort Index: (1/3)
            External Sort Progress: 3999999/4308303 92%",
    "progress" : {
      "done" : 3999999,
      "total" : 4308303
    },
    "numYields" : 0,
    "lockStats" : {
      "timeLockedMicros" : {},
      "timeAcquiringMicros" : {
        "r" : NumberLong(0),
        "w" : NumberLong(723)
      }
    }
  }
}
]
}

```

Msg字段描述构建的过程。注意，还有locks元素，它表示索引构建在stocks数据库上使用了一个锁。这意味着没有其他客户端可以同时读/写此数据库。如果在生产环境下，显然这个做法很糟糕，这也是为什么长时间构建索引让人头疼。我们来看看两个可行的解决方案。

后台索引

如果是在生产环境下无法停止数据库访问，就可以指定在后台构建索引。虽然构建索引还要占用写锁，但是此过程中允许其他用户读/写数据库。如果应用给予MongoDB的压力很大，后台索引构建就会影响性能，但是这些影响对于特定的环境是可以接受的。例如，如果知道构建索引可以在一个浏览最小的时期里完成，则此时后台索引或许是个不错的选择。

要使用后台方式构建索引，就需要在声明索引的时候指定{background: true}参数。之前的索引可用如下方式构建：

```
db.values.createIndex({open: 1, close: 1}, {background: true})
```

离线索引

后台构建索引可能还会对生产服务器造成无法接受的压力。如果这样，就需要离线构建索引了。这样通常需要离线复制一个新的服务器节点，然后在此服务器上创建索引，并且允许此服务器复制主服务器数据。一旦更新完毕，就可以把此服务器作为主服务器，然后采用第二台离线服务器构建其索引的版本。这个策略假设复制oplog日志足够大，以避免脱机服务器在索引构建过程中丢失数据。第11章详细介绍了复制集，应该可以帮助大家顺利完成计划迁移工作。

备份

因为难以构建索引，所以有时候我们想备份它们。不幸的是，不是所有的备份方法都支持索

引。例如，你也许尝试使用mongodump和mongorestore，但是这些构建只会保存集合和索引。这意味着运行mongorestore时，所有集合定义的索引只要备份过，都会重新创建恢复。通常，如果数据集很大，构建这些索引花费的时间可能无法接受。

所以，如果想备份索引，可能想单独备份MongoDB数据文件。关于这个问题的更多内容，也可以参考第13章的深入介绍。

碎片整理

如果应用对于数据库执行大量更新和删除操作，可能会产生许多索引碎片。B-树也会自己调整一些空间，但是这对于高删除空间还是不足的。索引碎片最大的问题是实际的空间远远大于数据需要的空间。索引碎片会导致使用更多的内存空间。这时候，我们可以考虑重建索引了。

我们可以通过删除并运行reIndex命令重新创建索引来实现，它会为集合重新创建所有的索引：

```
db.values.reIndex();
```

重建索引要格外小心：此命令在重建期间会占用写入锁，导致MongoDB实例无法使用。重建索引最好脱机进行，正如之前描述的。注意，第10章介绍的compact命令也可以为集合重新构建索引。

我们已经讨论了如何创建和管理索引。尽管掌握了这些知识，但是有时候你还会发现查询不够快。这个问题可能会在我们添加数据、流量或者新的查询时出现。让我们来学习如何对查询进行优化，以便加速查询并改善其性能。

8.3 查询优化

Query optimization

查询优化是指找出慢速查询，发现问题并解决问题的过程。本节里，我们将会逐步看一下查询优化的过程，这样在结束学习的时候就会掌握一整套处理MongoDB查询优化的方法。

在深入学习之前，我们必须声明，这里讲解的技巧、方法无法解决所有遇到的性能问题。因为导致慢速查询的原因各种各样：糟糕的应用程序架构设计、不正确的数据模型、硬件配置不足等，这些都需要花费时间来解决。这里我们将会介绍如何通过重构查询语句和索引来优化查询性能。我们也会介绍一些其他方法。当这些方法不能解决问题时，我们也介绍其他的途径。

8.3.1 找出慢速查询

如果感觉基于MongoDB应用变得迟缓了，就可以开始监控分析查询了。任何应用程序的设计原则都应该包括查询审计，而MongoDB的此项工作非常简单。虽然每个应用的需求各种各样，但可以假设绝大部分查询的时间不会超过100 ms。MongoDB日志器会打印索引查询时间超过100 ms的操作。所以，这些日志是我们分析慢速查询的第一个入口。

到目前为止，所有我们操作过的数据集都还没有大到要耗费100 ms进行查询操作。下面的例子，我们将会使用包含NASDAQ（纳斯达克股票）每日总结数据的数据集来实验。如果要动手实战，就需要自己下载到本地，然后导入本地数据库中。首先要在<http://mng.bz/ii49>下载数据库，然后解压到临时文件夹。会看到下面的输出信息：

```
$ unzip stocks.zip
Archive: stocks.zip
  creating: dump/stocks/
  inflating: dump/stocks/system.indexes.bson
  inflating: dump/stocks/values.bson
```

最后，启动mongod数据库实例，恢复备份文件：

```
$ mongorestore -d stocks dump/stocks
```

这个过程可能要花费一些时间。我们可能会在开始和结束的时候收到一些警告信息。不要担心这些。股票数据很大，而且易于处理。对于纳斯达克股票交易数据的子集，从1983年开始的25年，每天的最高价、最低价、收盘价的记录都会有一个文档。基于此集中的数据量和大小，很容易生成日志警告。尝试查询谷歌股票价格第一条记录：

```
use stocks
db.values.find({"stock_symbol": "GOOG"}).sort({date: -1}).limit(1)
```

你会注意到，这个命令会花费一些时间来执行，如果检查MongoDB日志，就会看到慢速查询的警告。以下是一个MongoDB 2.6里输出的信息：

```
Mon Sep 30 21:48:58.066 [conn20] query stocks.values query: { query: {
  stock_symbol: "GOOG" }, orderby: { date: -1.0 } }
  ntoreturn:1 ntoskip:0 nscanned:4308303 scanAndOrder:1 keyUpdates:0
  numYields: 3 locks(micros) r:4399440
  nreturned:1 reslen:194 4011ms
```

MongoDB v3.0在另外一个集合使用下面的命令，可以产生相似的日志信息：

```
2015-09-11T21:17:15.414+0300 I COMMAND [conn99] command green.$cmd command:
insert { insert: "system.indexes", documents: [ { _id:
ObjectId('55f31aab9a50479be0a7dcd7'), ns: "green.users", key: {
addresses.zip: 1.0 }, name: "zip" } ], ordered: false } keyUpdates:0
writeConflicts:0 numYields:0 reslen:40 locks:{ Global: { acquireCount: { r:
1, w: 1 } }, MMAPV1Journal: { acquireCount: { w: 9 } }, Database: {
acquireCount: { W: 1 } }, Collection: { acquireCount: { W: 1 } }, Metadata: {
acquireCount: { W: 5 } } } 102ms
```

这里有许多信息，当我们讨论`explain()`时会再详细介绍。现在如果仔细阅读信息，应该可以抽出最重要的信息，那就是`stocks.values`上的查询。这个查询选择器包含对于`stock_symbol`的匹配，执行了一个排序；而且，也许最重要的是查询居然耗费了4 s(4011 ms)。执行时间可能和你的计算机配置有关系。

这样的警告必须处理。产生了如此严重的错误，以致大家必须查看MongoDB日志。可以使用`grep`简单地实现查询工作：

```
grep -E '[0-9]+ms' mongod.log
```

100 ms是很高的门槛，在启动MongoDB时，我们可以使用`-slowms`参数设置、指定。如果要记录超过50 ms的操作，可以在启动`mongod`时使用`--slowms 50`参数。

当然，使用`grep`查询日志信息不够详细。我们也可以使用MongoDB日志来检查慢速查询，但是过程比较粗糙，应该作为生产环境下的保留检查方式。要在慢速查询变成问题之前找出慢速查询，我们需要精确的工具。MongoDB内置了查询分析器来帮助完成这个工作。

使用 PROFILER 分析器

对于慢速查询的分析，不能找到比内置分析器更好的工具了。默认情况下禁用了这个工具，所以我们开始启用它。在MongoDB shell里输入以下命令：

```
use stocks
db.setProfilingLevel(2)
```

首先，选择要监控的数据库，分析的范围通常是某个数据库。我们可以把分析级别设置为2。这是最详细的级别，它会告诉分析器记录每个读/写操作。其他参数也可以使用。记录慢速操作耗时超时(100 ms)，要设置监控级别为1。要禁用分析器，可以设置为0。分析器只会记录耗时超时的操作，传递毫秒作为第二个参数，如下所示：

```
use stocks
db.setProfilingLevel(1, 50)
```

一旦启用了监控，就可以尝试一下查询了。我们来运行另外一个股票数据库上的查询。要查询数据集里的最高收盘价：

```
db.values.find({}).sort({close: -1}).limit(1)
```

监控结果

监控结果保存到一个特殊的盖子集合`system.profile`里，它存储在执行`setProfilingLevel`命令的数据库中。

回忆下，盖子集合具有固定的大小，而且数据写入的方式很特殊，一旦达到最大数量，新文档就会取代旧的文档。`system.profile`集合分配了128 KB空间，因此要确保监控分析数据不会消耗太多的资源。

我们也可以作为盖子集合查询system.profile。例如，可以查找所有耗时超过150 ms的操作，如下所示：

```
db.system.profile.find({millis: {$gt: 150}})
```

因为盖子集合维护了自然的插入顺序，所以，可以使用\$natural操作符来排序，让最近的结果可以首先显示出来：

```
db.system.profile.find().sort({$natural: -1}).limit(5).pretty()
```

返回刚才的查询，看到结果集中的入口，如下所示：

```
{
  "op" : "query",
  "ns" : "stocks.values",
  "query" : {
    "query" : { },
    "orderby" : {
      "close" : -1
    }
  },
  "ntoreturn" : 1,
  "ntoskip" : 0,
  "nscanned" : 4308303,
  "scanAndOrder" : true,
  "keyUpdates" : 0,
  "numYield" : 3,
  "lockStats" : {
    "timeLockedMicros" : {
      "r" : NumberLong(12868747),
      "w" : NumberLong(0)
    },
    "timeAcquiringMicros" : {
      "r" : NumberLong(1838271),
      "w" : NumberLong(5)
    }
  },
  "nreturned" : 1,
  "responseLength" : 194,
  "millis" : 11030,
  "ts" : ISODate("2013-09-30T06:44:40.988Z"),
  "client" : "127.0.0.1",
  "allUsers" : [ ],
  "user" : ""
}
```

集合名字

扫描文档的数量

返回文档的数量

毫秒应答时间

这是另外一个昂贵的查询：花费11s！除了花费的时间，我们还获得了MongoDB日志里的所有查询警告信息，这对于下一节里更深入的分析已经足够了。

在继续深入之前，我们再来顺序强调一下监控策略：

- 使用监控分析器的良好方式是使用粗略设置和向下工作模式启动它。首先确保没有超过100 ms的查询，然后继续到75 ms查询，以此类推。

- 当启用监控分析器后，我们想要监控所有的应用程序操作，这意味着测试所有应用程序的读取和写入操作。
- 要彻底分析问题，这些读/写操作应该在真实的条件下执行，当数据大小、查询负载、硬件与生产环境一样时，查询效果最好。

查询分析器非常有用，但是要获取最大的收益，需要我们有有条不紊地操作。最好的做法是在开发阶段就找出一些慢速的查询，因为在生产阶段发现问题、解决问题的成本更高。

作为参考，下面的输出监控结果是基于MongoDB v3.0.6版本：

```
> db.system.profile.find().limit(1).pretty()
```

```
{
  "op" : "query",
  "ns" : "products.system.profile",
  "query" : {
  },
  "ntoreturn" : 0,
  "ntoskip" : 0,
  "nscanned" : 0,
  "nscannedObjects" : 0,
  "keyUpdates" : 0,
  "writeConflicts" : 0,
  "numYield" : 0,
  "locks" : {
    "Global" : {
      "acquireCount" : {
        "r" : NumberLong(2)
      }
    },
    "MMAPV1Journal" : {
      "acquireCount" : {
        "r" : NumberLong(1)
      }
    },
    "Database" : {
      "acquireCount" : {
        "r" : NumberLong(1)
      }
    },
    "Collection" : {
      "acquireCount" : {
        "R" : NumberLong(1)
      }
    }
  },
  "nreturned" : 0,
  "responseLength" : 20,
  "millis" : 1,
  "execStats" : {
    "stage" : "COLLSCAN",
    "filter" : {
      "$and" : [ ]
    }
  },
}
```

```

    "nReturned" : 0,
    "executionTimeMillisEstimate" : 0,
    "works" : 2,
    "advanced" : 0,
    "needTime" : 1,
    "needFetch" : 0,
    "saveState" : 0,
    "restoreState" : 0,
    "isEOF" : 1,
    "invalidates" : 0,
    "direction" : "forward",
    "docsExamined" : 0
  },
  "ts" : ISODate("2015-09-11T18:52:08.847Z"),
  "client" : "127.0.0.1",
  "allUsers" : [ ],
  "user" : ""
}

```

8.3.2 检查慢速查询

使用MongoDB的查询分析器找出慢速查询非常容易。但是要找出其原因就比较棘手了，就需要一些探测分析工作。正如之前提到的，查询速度慢的原因有许多个。如果幸运，添加索引就可以解决问题。在更多的情况下，可能必须重新组织索引，重构数据模型，或者升级硬件。但是通常我们应该先从最简单的方法开始，这也是现在我们要开始做的事情。

在最简单的情况下，缺少索引、不恰当的索引低于期望的查询都会遇到这些问题。我们可以通过在这些查询上运行explain命令来找出原因。现在我们就来看看具体如何操作。

使用并理解 explain()

MongoDB的explain命令提供了关于查询语句的详细信息。直接进入实例，看看可以从上一个部分运的最后一个查询解释中收集哪些信息。要在shell里运行explain，只需要在语句后追加explain()方法即可：

```

db.values.find({}).sort({close: -1}).limit(1).explain()
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 4308303,
  "nscanned" : 4308303,
  "nscannedObjectsAllPlans" : 4308303,
  "nscannedAllPlans" : 4308303,
  "scanAndOrder" : true,
  "indexOnly" : false,
  "nYields" : 4,
  "nChunkSkips" : 0,
  "millis" : 10927,
  "indexBounds" : { },

```

返回数量

扫描数量


```
"server" : "localhost:27017"
```

millis字段显示查询耗费了11 s^[1]，而且没有明显的原因。看一下nscanned的值：它显示查询引擎扫描了4 308 303个文档。现在，我们快速在此集合上运行count函数：

```
db.values.count()  
4308303
```

查询扫描的文档和集合中总的文档数量相同，所以我们执行了一个完整的集合扫描。如果希望返回集合里的每个文档，这还说得过去。但是如果返回一个文档，正如解释值n所示，这就表示有问题。

此外，如果添加了更多的文档，则全集合扫描会变得更昂贵。通常来说，我们希望n和nscanned尽量接近。当进行集合扫描时，这是几乎从来没有的情况。cursor字段告诉我们，一直在使用BasicCursor，它表示在扫描集合而不是索引。如果使用的是索引，那个值就是BTreeCursor了。

第二个参数scanAndOrder字段具体介绍了查询缓慢的原因。这个指示器会在查询优化器无法使用索引返回排序集合时出现。因此，此时不仅仅要查询引擎要扫描集合，还必须手动排序结果集合。

之前explain()命令的输出结果来自旧的MongoDB版本。这个例子的explain()命令的输出结果来自MongoDB v3.0.6：

```
> db.inventory.find({}).sort({"quantity": -1}).limit(1).  
  explain("executionStats")  
{  
  "queryPlanner" : {  
    "plannerVersion" : 1,  
    "namespace" : "tutorial.inventory",  
    "indexFilterSet" : false,  
    "parsedQuery" : {  
      "$and" : [ ]  
    },  
    "winningPlan" : {  
      "stage" : "SORT",  
      "sortPattern" : {  
        "quantity" : -1  
      },  
      "limitAmount" : 1,  
      "inputStage" : {  
        "stage" : "COLLSCAN",  
        "filter" : {  
          "$and" : [ ]  
        },  
        "direction" : "forward"  
      },  
      "rejectedPlans" : [ ]  
    }  
  },  
  "rejectedPlans" : [ ]  
}
```

^[1]看起来不多，但当用户访问网站、加载网页、在后台进行数据库查询时，11s太久啦。

```

},
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 1,
  "executionTimeMillis" : 0,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 11,
  "executionStages" : {
    "stage" : "SORT",
    "nReturned" : 1,
    "executionTimeMillisEstimate" : 0,
    "works" : 16,
    "advanced" : 1,
    "needTime" : 13,
    "needFetch" : 0,
    "saveState" : 0,
    "restoreState" : 0,
    "isEOF" : 1,
    "invalidates" : 0,
    "sortPattern" : {
      "quantity" : -1
    },
    "memUsage" : 72,
    "memLimit" : 33554432,
    "limitAmount" : 1,
    "inputStage" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "$and" : [ ]
      },
      "nReturned" : 11,
      "executionTimeMillisEstimate" : 0,
      "works" : 13,
      "advanced" : 11,
      "needTime" : 1,
      "needFetch" : 0,
      "saveState" : 0,
      "restoreState" : 0,
      "isEOF" : 1,
      "invalidates" : 0,
      "direction" : "forward",
      "docsExamined" : 11
    }
  },
  "serverInfo" : {
    "host" : "rMacBook.local",
    "port" : 27017,
    "version" : "3.0.6",
    "gitVersion" : "nogitversion"
  },
  "ok" : 1
}

```

当使用executionStats 参数时, explain() 命令显示了更多的信息。

添加索引并查询

糟糕的性能是无法接受的,但是幸运的是解决这个问题也非常简单。所有要做的就是为close字段建立索引。继续如下命令:

```
db.values.createIndex({close: 1})
```

注意,构建索引可能要花费一些时间。一旦完成,可以再次尝试查询:

```
db.values.find({}).sort({close: -1}).limit(1).explain()
```

```
{
  "cursor" : "BtreeCursor close_1 reverse",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
    "name" : [
      [
        {
          "$maxElement" : 1
        },
        {
          "$minElement" : 1
        }
      ]
    ]
  },
  "server" : "localhost:27017"
}
```

发生了翻天覆地的变化!查询花费了不到1 ms的时间。我们可以看到cursor使用BtreeCursor在名为close_1的索引上进行搜索,使用反序迭代。在indexBounds字段里,会看到特殊的值\$maxElement和\$minElement。它们表示查询范围的最大值和最小值。此时,查询优化器会搜寻直到B-树最右边的最大值,然后往回搜索。因为我们已经指定了限制为1,所以查询会在找到maxElement后立即完成。当然,索引会按照顺序保留所有的入口,因此,没有必要再使用scanAndOrder排序。

相似的是, MongoDB 3.0.6输出结果展示了改进后的执行时间,以及检查过的文档数量:

```
> db.inventory.find({}).sort({"quantity": -
  1}).limit(1).explain("executionStats")
{
  "queryPlanner": {
    "plannerVersion" : 1,
    "namespace" : "tutorial.inventory",
```

```

    "indexFilterSet" : false,
    "parsedQuery" : {
      "$and" : [ ]
    },
    "winningPlan" : {
      "stage" : "LIMIT",
      "limitAmount" : 0,
      "inputStage" : {
        "stage" : "FETCH",
        "keyPattern" : {
          "quantity" : 1
        },
        "indexName" : "quantity_1",
        "isMultiKey" : false,
        "direction" : "backward",
        "indexBounds" : {
          "quantity" : [
            "[MaxKey, MinKey]"
          ]
        }
      },
      "rejectedPlans" : [ ]
    },
    "executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 1,
      "executionTimeMillis" : 0,
      "totalKeysExamined" : 1,
      "totalDocsExamined" : 1,
      "executionStages" : {
        "stage" : "LIMIT",
        "nReturned" : 1,
        "executionTimeMillisEstimate" : 0,
        "works" : 2,
        "advanced" : 1,
        "needTime" : 0,
        "needFetch" : 0,
        "saveState" : 0,
        "restoreState" : 0,
        "isEOF" : 1,
        "invalidates" : 0,
        "limitAmount" : 0,
        "inputStage" : {
          "stage" : "FETCH",
          "nReturned" : 1,
          "executionTimeMillisEstimate" : 0,
          "works" : 1,
          "advanced" : 1,
          "needTime" : 0,
          "needFetch" : 0,
          "saveState" : 0,
          "restoreState" : 0,
          "isEOF" : 0,
          "invalidates" : 0,
          "docsExamined" : 1,

```

```

    "alreadyHasObj" : 0,
    "inputStage" : {
      "stage" : "IXSCAN",
      "nReturned" : 1,
      "executionTimeMillisEstimate" : 0,
      "works" : 1,
      "advanced" : 1,
      "needTime" : 0,
      "needFetch" : 0,
      "saveState" : 0,
      "restoreState" : 0,
      "isEOF" : 0,
      "invalidates" : 0,
      "keyPattern" : {
        "quantity" : 1
      },
      "indexName" : "quantity_1",
      "isMultiKey" : false,
      "direction" : "backward",
      "indexBounds" : {
        "quantity" : [
          "[MaxKey, MinKey]"
        ]
      },
      "keysExamined" : 1,
      "dupsTested" : 0,
      "dupsDropped" : 0,
      "seenInvalidated" : 0,
      "matchTested" : 0
    }
  }
},
"serverInfo" : {
  "host" : "rMacBook.local",
  "port" : 27017,
  "version" : "3.0.6",
  "gitVersion" : "nogitversion"
},
"ok" : 1
}

```

展示MongoDB v2.x和MongoDB v3.0版本结果的原因是给大家作为参考。

使用索引键

如果在查询选择器里使用索引键会看到不同的输出结果。看一下查询选择收盘价大于500的查询执行计划：

```

> db.values.find({close: {$gt: 500}}).explain()
{
  "cursor" : "BtreeCursor close_1",
  "isMultiKey" : false,
  "n" : 309,
  "nscannedObjects" : 309,

```

```

"nscanned" : 309,
"nscannedObjectsAllPlans" : 309,
"nscannedAllPlans" : 309,
"scanAndOrder" : false,
"indexOnly" : false,
"nYields" : 0,
"nChunkSkips" : 0,
"millis" : 1,
"indexBounds" : {
  "close" : [
    [
      500,
      1.7976931348623157e+308
    ]
  ]
},
"server" : "localhost:27017"
}

```

扫描和要返回的文档数量一样（n和nscanned一样），这是理想的。但是要注意指定的索引边界的差别。边界是实际的值而不是\$maxElement和\$minElement键。最低是500，最高是无穷大。查询的这些值必须有相同的数据类型：我们在数值上查询，意味着边界就是数值；如果查询的是字符串，那么边界应该就是字符串^[1]。

通常，MongoDB 3.0相似的查询输出结果如下所示：

```

> db.inventory.find({"quantity":{"$gt":150}}).limit(1).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "tutorial.inventory",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "quantity" : {
        "$gt" : 150
      }
    },
    "winningPlan" : {
      "stage" : "LIMIT",
      "limitAmount" : 0,
      "inputStage" : {
        "stage" : "FETCH",
        "inputStage" : {
          "stage" : "IXSCAN",
          "keyPattern" : {
            "quantity" : 1
          },
          "indexName" : "quantity_1",
          "isMultiKey" : false,
          "direction" : "forward",
          "indexBounds" : {
            "quantity" : [

```

^[1]如果这个不对，就回忆一下索引可以包含多个数据类型的键值。因此，查询结果将会受查询中的数据类型限制。


```

        "(150.0, inf.0]"
    ]
  }
}
},
"rejectedPlans" : [ ]
},
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 1,
  "executionTimeMillis" : 0,
  "totalKeysExamined" : 1,
  "totalDocsExamined" : 1,
  "executionStages" : {
    "stage" : "LIMIT",
    "nReturned" : 1,
    "executionTimeMillisEstimate" : 0,
    "works" : 2,
    "advanced" : 1,
    "needTime" : 0,
    "needFetch" : 0,
    "saveState" : 0,
    "restoreState" : 0,
    "isEOF" : 1,
    "invalidates" : 0,
    "limitAmount" : 0,
    "inputStage" : {
      "stage" : "FETCH",
      "nReturned" : 1,
      "executionTimeMillisEstimate" : 0,
      "works" : 1,
      "advanced" : 1,
      "needTime" : 0,
      "needFetch" : 0,
      "saveState" : 0,
      "restoreState" : 0,
      "isEOF" : 0,
      "invalidates" : 0,
      "docsExamined" : 1,
      "alreadyHasObj" : 0,
      "inputStage" : {
        "stage" : "IXSCAN",
        "nReturned" : 1,
        "executionTimeMillisEstimate" : 0,
        "works" : 1,
        "advanced" : 1,
        "needTime" : 0,
        "needFetch" : 0,
        "saveState" : 0,
        "restoreState" : 0,
        "isEOF" : 0,
        "invalidates" : 0,
        "keyPattern" : {
          "quantity" : 1
        },
        "indexName" : "quantity_1",

```

```

        "isMultiKey" : false,
        "direction" : "forward",
        "indexBounds" : {
          "quantity" : [
            "(150.0, inf.0]"
          ]
        },
        "keysExamined" : 1,
        "dupsTested" : 0,
        "dupsDropped" : 0,
        "seenInvalidated" : 0,
        "matchTested" : 0
      }
    }
  },
  "serverInfo" : {
    "host" : "rMacBook.local",
    "port" : 27017,
    "version" : "3.0.6",
    "gitVersion" : "nogitversion"
  },
  "ok" : 1
}

```

在继续之前，可以针对不同版本的MongoDB查询运行explain()命令，注意n和nscanned的差别，还有totalDocsExamined和nReturned的不同。优化MongoDB 2.x版本的查询，通常指的是尽可能让nscanned的值变小，但是每个结果必须被扫描，所以nscanned充不能比n低，查询返回结果的数量。在MongoDB 3.0中，nReturned指的是匹配查询和返回的文档数量。totalDocsExamined指的是MongoDB扫描的文档数量。最后，totalKeysExamined显示的是MongoDB扫描索引的入口数量。

MONGODB 的查询优化器

对于给定的查询，查询优化器决定哪个索引的效率最高。要为查询选择最理性的索引，查询优化器使用了一系列相当简单的规则：

- (1) 避免scanAndOrder。如果查询包含排序，则尝试使用索引排序。
- (2) 使用有用的索引约束满足所有字段——尝试为查询选择器中的字段使用索引。
- (3) 如果查询包含范围或者排序，则选择最后一个key使用的索引来帮助处理范围和排序。

如果对于任意的索引这些条件都可以满足，那么所有索引就会是最佳的并且可以使用。如果多个索引都最佳，就任意选择其中一个最佳索引。这里有个教训：如果可以为查询构建最佳索引，就可以大大简化优化器的工作。因此请尽量做到。

我们来看一个完美满足查询的索引（还有查询优化器）。回到股票代码数据库，假设我们想查询谷歌Google收盘价大于200的所有股票数据。

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}})
```

如果使用`.explain()`分析此查询, 就会看到`n`是730, 但是`nscanned`是5299。之前在`close`字段上创建的索引起了很大作用, 但是本次查询的最佳索引包含了键值并且把`close`放到最后以允许范围查询。

```
db.values.createIndex({stock_symbol: 1, close: 1})
```

如果运行查询, 会看到两个键值都会被使用, 而且索引边界正如我们期望的, 如下所示:

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}}).explain()
```

```
{
  "cursor" : "BtreeCursor stock_symbol_1_close_1",
  "isMultiKey" : false,
  "n" : 730,
  "nscannedObjects" : 730,
  "nscanned" : 730,
  "nscannedObjectsAllPlans" : 730,
  "nscannedAllPlans" : 730,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 2,
  "indexBounds" : {
    "stock_symbol" : [
      [
        "GOOG",
        "GOOG"
      ],
      "close" : [
        [
          200,
          1.7976931348623157e+308
        ]
      ]
    ],
    "server" : "localhost:27017"
  }
}
```

这是优化的`explain`针对本次查询的输出结果: `n`和`nscanned`一样。但是, 现在思考一下没有完美的索引服务查询的情况。例如, 在`{stock_symbol: 1, close: 1}`上没有索引, 但是在每个字段上有独立的索引。使用`getIndexKeys()`来列举索引, 就会看到如下信息:

```
db.values.getIndexKeys()
```

```
[
  {
    "_id" : 1
  },
  {
    "close" : 1
  },
  {
    "stock_symbol" : 1
  }
]
```

```

    }
  }
}

```

因为查询包含stock_symbol和close键，没有显示使用索引。这也是查询优化器介入的原因，而且启发式做法比我们想的更简单。它只是根据nscanned的值进行决策。换句话说，优化器选择扫描索引入口数量最少的索引。当查询第一次运行时，每个索引的查询计划可能都可以高效地满足查询需求。优化器并行运行每个计划^[1]。通常，带有nscanned最低值的执行计划脱颖而出，成为最终赢家。但是优化器很少情况下会选择全集合扫描作为查询的优化方案。优化器会暂停长时间运行的计划，并且保持优胜者作为以后使用。

下面是MongoDB 3.0的输出结果，使用了更少的测试数据集合：

```

> db.inventory.find({"quantity": 500,
  "type": "toys"}).limit(1).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "tutorial.inventory",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "$and" : [
        {
          "quantity" : {
            "$eq" : 500
          }
        },
        {
          "type" : {
            "$eq" : "toys"
          }
        }
      ]
    },
    "winningPlan" : {
      "stage" : "LIMIT",
      "limitAmount" : 0,
      "inputStage" : {
        "stage" : "KEEP_MUTATIONS",
        "inputStage" : {
          "stage" : "FETCH",
          "filter" : {
            "type" : {
              "$eq" : "toys"
            }
          }
        },
        "inputStage" : {
          "stage" : "IXSCAN",
          "keyPattern" : {
            "quantity" : 1
          },
          "indexName" : "quantity_1",
          "isMultiKey" : false,

```

^[1]从技术上说，计划是交错执行的。

```

        "direction" : "forward",
        "indexBounds" : {
          "quantity" : [
            "[500.0, 500.0]"
          ]
        }
      },
      "rejectedPlans" : [ ]
    },
    "executionStats" : {
      "executionSuccess" : true,
      "nReturned" : 1,
      "executionTimeMillis" : 1,
      "totalKeysExamined" : 2,
      "totalDocsExamined" : 2,
      "executionStages" : {
        "stage" : "LIMIT",
        "nReturned" : 1,
        "executionTimeMillisEstimate" : 0,
        "works" : 3,
        "advanced" : 1,
        "needTime" : 1,
        "needFetch" : 0,
        "saveState" : 0,
        "restoreState" : 0,
        "isEOF" : 1,
        "invalidates" : 0,
        "limitAmount" : 0,
        "inputStage" : {
          "stage" : "KEEP_MUTATIONS",
          "nReturned" : 1,
          "executionTimeMillisEstimate" : 0,
          "works" : 2,
          "advanced" : 1,
          "needTime" : 1,
          "needFetch" : 0,
          "saveState" : 0,
          "restoreState" : 0,
          "isEOF" : 0,
          "invalidates" : 0,
          "inputStage" : {
            "stage" : "FETCH",
            "filter" : {
              "type" : {
                "$eq" : "toys"
              }
            }
          },
          "nReturned" : 1,
          "executionTimeMillisEstimate" : 0,
          "works" : 2,
          "advanced" : 1,
          "needTime" : 1,
          "needFetch" : 0,
          "saveState" : 0,

```

```

"restoreState" : 0,
"isEOF" : 1,
"invalidates" : 0,
"docsExamined" : 2,
"alreadyHasObj" : 0,
"inputStage" : {
  "stage" : "IXSCAN",
  "nReturned" : 2,
  "executionTimeMillisEstimate" : 0,
  "works" : 2,
  "advanced" : 2,
  "needTime" : 0,
  "needFetch" : 0,
  "saveState" : 0,
  "restoreState" : 0,
  "isEOF" : 1,
  "invalidates" : 0,
  "keyPattern" : {
    "quantity" : 1
  },
  "indexName" : "quantity_1",
  "isMultiKey" : false,
  "direction" : "forward",
  "indexBounds" : {
    "quantity" : [
      "[500.0, 500.0]"
    ]
  },
  "keysExamined" : 2,
  "dupsTested" : 0,
  "dupsDropped" : 0,
  "seenInvalidated" : 0,
  "matchTested" : 0
},
},
},
"serverInfo" : {
  "host" : "rMacBook.local",
  "port" : 27017,
  "version" : "3.0.6",
  "gitVersion" : "nogitversion"
},
"ok" : 1
}

```

前面介绍的查询检查了2个文档后返回了期望的文档。现在是时候创建一个包含两个字段的索引了:

```

> db.inventory.createIndex( { quantity: 1, type: 1 } )
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 2,
  "numIndexesAfter" : 3,

```



```

    "ok" : 1
  }
}

```

现在我们要回到之前的查询:

```

> db.inventory.find({"quantity": 500,
  "type": "toys"}).limit(1).explain("executionStats")
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "tutorial.inventory",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "$and" : [
        {
          "quantity" : {
            "$eq" : 500
          }
        },
        {
          "type" : {
            "$eq" : "toys"
          }
        }
      ]
    },
    "winningPlan" : {
      "stage" : "LIMIT",
      "limitAmount" : 0,
      "inputStage" : {
        "stage" : "FETCH",
        "inputStage" : {
          "stage" : "IXSCAN",
          "keyPattern" : {
            "quantity" : 1,
            "type" : 1
          },
          "indexName" : "quantity_1_type_1",
          "isMultiKey" : false,
          "direction" : "forward",
          "indexBounds" : {
            "quantity" : [
              "[500.0, 500.0]"
            ],
            "type" : [
              "[\"toys\", \"toys\"]"
            ]
          }
        }
      },
      "rejectedPlans" : [
        {
          "stage" : "LIMIT",
          "limitAmount" : 1,
          "inputStage" : {
            "stage" : "KEEP_MUTATIONS",

```

```

        "inputStage" : {
            "stage" : "FETCH",
            "filter" : {
                "type" : {
                    "$eq" : "toys"
                }
            },
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "quantity" : 1
                },
                "indexName" : "quantity_1",
                "isMultiKey" : false,
                "direction" : "forward",
                "indexBounds" : {
                    "quantity" : [
                        "[500.0, 500.0]"
                    ]
                }
            }
        },
        "executionStats" : {
            "executionSuccess" : true,
            "nReturned" : 1,
            "executionTimeMillis" : 1,
            "totalKeysExamined" : 1,
            "totalDocsExamined" : 1,
            "executionStages" : {
                "stage" : "LIMIT",
                "nReturned" : 1,
                "executionTimeMillisEstimate" : 0,
                "works" : 2,
                "advanced" : 1,
                "needTime" : 0,
                "needFetch" : 0,
                "saveState" : 0,
                "restoreState" : 0,
                "isEOF" : 1,
                "invalidates" : 0,
                "limitAmount" : 0,
                "inputStage" : {
                    "stage" : "FETCH",
                    "nReturned" : 1,
                    "executionTimeMillisEstimate" : 0,
                    "works" : 1,
                    "advanced" : 1,
                    "needTime" : 0,
                    "needFetch" : 0,
                    "saveState" : 0,
                    "restoreState" : 0,
                    "isEOF" : 1,
                    "invalidates" : 0,

```

```

        "docsExamined" : 1,
        "alreadyHasObj" : 0,
        "inputStage" : {
            "stage" : "IXSCAN",
            "nReturned" : 1,
            "executionTimeMillisEstimate" : 0,
            "works" : 1,
            "advanced" : 1,
            "needTime" : 0,
            "needFetch" : 0,
            "saveState" : 0,
            "restoreState" : 0,
            "isEOF" : 1,
            "invalidates" : 0,
            "keyPattern" : {
                "quantity" : 1,
                "type" : 1
            },
            "indexName" : "quantity_1_type_1",
            "isMultiKey" : false,
            "direction" : "forward",
            "indexBounds" : {
                "quantity" : [
                    "[500.0, 500.0]"
                ],
                "type" : [
                    "[\"toys\", \"toys\"]"
                ]
            },
            "keysExamined" : 1,
            "dupsTested" : 0,
            "dupsDropped" : 0,
            "seenInvalidated" : 0,
            "matchTested" : 0
        }
    }
},
"serverInfo" : {
    "host" : "rMacBook.local",
    "port" : 27017,
    "version" : "3.0.6",
    "gitVersion" : "nogitversion"
},
"ok" : 1
}

```

这次只有一个被检查的文档返回了一个文档。这意味着新的索引在优化查询中起了作用。

显示查询计划和 HINT()

我们可以通过运行 `explain()` 命令来查看这个过程信息。首先，删除 `{stock_symbol: 1, close: 1}` 上的复合索引，然后在每个字段上建立单独的索引：

```
db.values.dropIndex("stock_symbol_1_close_1")
db.values.createIndex({stock_symbol: 1})
db.values.createIndex ({close: 1})
```

然后传递true给explain()方法，它包含查询优化器尝试的计划列表。我们可以在列表8.2看到输出信息。当使用MongoDB 3.0后，可能的模式是queryPlanner、executionStats以及allPlansExecution三种。为了兼容早期版本的cursor.explain()，MongoDB 3.0把allPlansExecution设置为true，把queryPlanner设置为false。

列表8.2 使用explain(true)查看查询计划。

```
db.values.find({stock_symbol: "GOOG", close: {$gt: 200}}).explain(true),
{
  "cursor" : "BtreeCursor stock_symbol_1",
  "isMultiKey" : false,
  "n" : 730,
  "nscannedObjects" : 894,
  "nscanned" : 894,
  "nscannedObjectsAllPlans" : 1097,
  "nscannedAllPlans" : 1097,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 4,
  "indexBounds" : {
    "stock_symbol" : [
      "GOOG",
      "GOOG"
    ]
  },
  "allPlans" : [
    {
      "cursor" : "BtreeCursor close_1",
      "n" : 0,
      "nscannedObjects" : 102,
      "nscanned" : 102,
      "indexBounds" : {
        "close" : [
          200,
          1.7976931348623157e+308
        ]
      }
    }
  ],
  "cursor" : "BtreeCursor stock_symbol_1",
  "n" : 730,
  "nscannedObjects" : 894,
  "nscanned" : 894,
  "indexBounds" : {
    "stock_symbol" : [
      "GOOG",
```

扫描的文档

查询时间

因为这是基于相等性查询

索引边界相等

尝试查询计划的数组

```

        "GOOG"
    ]
}
{
    "cursor": "BasicCursor",
    "n": 0,
    "nscannedObjects": 101,
    "nscanned": 101,
    "indexBounds": { }
}
],
"server": "localhost:27017"
}

```

我们看到查询计划选择{stock_symbol: 1}的索引来支持查询。再往下，allPlans的键指向了包含2个查询计划的列表：一个在索引{close: 1}上，一个是使用BasicCursor扫描的集合。MongoDB把这个列表叫做rejectedPlans。

为什么优化器拒绝集合扫描很好理解，但是为什么{close: 1}的索引不好，可能不太好明白。我们可以使用hint()来查看具体信息。hint()强制查询优化器来使用特定的索引：

```

query = {stock_symbol: "GOOG", close: {$gt: 200}}
db.values.find(query).hint({close: 1}).explain()
{
    "cursor": "BtreeCursor close_1",
    "isMultiKey": false,
    "n": 730,
    "nscannedObjects": 5299,
    "nscanned": 5299,
    "nscannedObjectsAllPlans": 5299,
    "nscannedAllPlans": 5299,
    "scanAndOrder": false,
    "indexOnly": false,
    "nYields": 0,
    "nChunkSkips": 0,
    "millis": 22,
    "indexBounds": {
        "close": [
            [
                200,
                1.7976931348623157e+308
            ]
        ]
    },
    "server": "localhost:27017"
}

```

看到nscanned的值为5 299。它比之前扫描的入口数量894大太多，而且花费的时间也难以忍受。

使用MongoDB 3.0运行相同的查询并分析输出结果信息的工作作为课后练习留给大家。

查询计划缓存

剩下的就是要理解查询优化器如何缓存和过期查询计划。毕竟，我们不希望查询优化器为查询并行执行所有的计划。

当找到一个成功的计划后，查询模式、nscanned的值以及指定的索引就会被记录下来。对于每个处理的查询，记录的数据结构看起来如下所示：

```
{
  pattern: {
    stock_symbol: 'equality',
    close: 'bound',
    index: {
      stock_symbol: 1
    },
    nscanned: 894
  }
}
```

查询模式记录了每个键的匹配类型。这里，我们查询对于stock_symbol的匹配记录（等于），而且在close范围（边界）匹配^[1]。当一个新的查询匹配这个模式时，就会使用对应的索引。

这个计划确实不会永久存在。优化器在下列事件发生后会自动过期计划：

- 集合中写入了100次。
- 集合新建或者删除了索引。
- 使用查询计划的查询做了比期望更多的工作。这里，衡量“更多的工作”用的是nscanned的值超过了缓存的nscanned的值至少是10。

在这些情况的最后，优化器会立即交互执行其他计划来寻找更高效的计划。随着大家花费时间来优化查询，很可能注意到几个查询的模式和索引一起工作得很好。下一节里，我们来深入介绍这几种模式。

如果你在使用MongoDB 3.0版本，则可以在下面的网站学习更多查询计划缓存的信息：<http://docs.mongodb.org/manual/reference/method/js-plan-cache/>。

8.3.3 查询模式

这里我们介绍几种常见的查询模式和一起使用的索引。本节的目标是帮助我们使用MongoDB索引来调优匹配应用程序的查询。

单键索引

要复习单键索引，就回忆一下之前在8.3.2节股票集合收盘号码{close: 1}上建立的索引。

^[1]如果感兴趣，如下三种类型的范围匹配都会保存：上限、下限和上-下限。查询模式也包含任意的排序说明。

这个索引可以用到下面的场景中。

(1) 精确匹配。无论是返回0个、1个，还是多个结果，都使用索引。在这些查询里使用精确匹配，返回所有收盘价为100的入口：

```
db.values.find({close: 100})
```

(2) 排序：在索引字段上排序。例如：

```
db.values.find({}).sort({close: 1})
```

当不使用查询选择器排序时，我们可能想要有个限制，除非我们真的想在整个集合上查询所有数据。

(3) 范围查询。范围查询可能在同一个字段上使用或者不使用排序。例如，查询所有的收盘价格大于或者等于100的股票：

```
db.values.find({close: {$gte: 100}})
```

如果在同一个键上添加排序语句，优化器仍然会使用相同的索引：

```
db.values.find({close: {$gte: 100}}).sort({close: 1})
```

复合索引

复合索引有点复杂，但是它们的使用类比为单键索引。主要要记住的是，复合索引可以高效地服务单键值范围或者每个查询排序。我们来看下三键索引，还是股票，在{close: 1, open: 1, date: 1}上的值。我们来看看一些可能的场景。

(1) 精确匹配。在第一个键上精确匹配，第一个和第二个键，或者第一、第二和第三个键使用指定的顺序：

```
db.values.find({close: 1})
db.values.find({close: 1, open: 1})
db.values.find({close: 1, open: 1, date: "1985-01-08"})
```

(2) 范围匹配。在任意集合最左边键（包括none）上的精确匹配，并且使用限制范围或者排序。因此，下面所有的查询对于三键查询都是理想情况：

```
db.values.find({}).sort({close: 1})
db.values.find({close: {$gt: 1}})
db.values.find({close: 100}).sort({open: 1})
db.values.find({close: 100, open: {$gt: 1}})
db.values.find({close: 1, open: 1.01, date: {$gt: "2005-01-01"}})
db.values.find({close: 1, open: 1.01}).sort({date: 1})
```

(3) 覆盖索引。如果你从来没有听过覆盖索引，那么一开始，这个词就有点措辞不当。覆盖索引不是一种索引，而是一个索引的特殊使用。特别是，如果需要的数据都驻留在索引里，索引可以说成覆盖了查询。覆盖索引也称索引查询，因为这些查询不需要访问具体的索引文档。这会大大改善查询性能。

在MongoDB里使用覆盖索引非常简单。简单选择驻留在单个索引里的字段集合，排除_id字段（这个字段不太可能使用到）。下面是个使用三键索引的例子：

```
db.values.find({close: 1}, {open: 1, close: 1, date: 1, _id: 0})
```

在早期版本的MongoDB里，`cursor.explain()`返回`indexOnly`字段，表示索引是否覆盖了查询，以及是否有具体的集合数据服务于查询。在MongoDB 3.0里，当索引覆盖了某个查询时，`explain`命令的结果包含`IXSCAN`阶段而不是`FETCH`阶段的后代，而且在`executionStats`里，`totalDocsExamined`的值是0。

8.4 总结

Summary

本章内容比较深，因为索引是公认的非常丰富、重要的主题。如果还有什么概念不够清晰，也没有关系，至少已经掌握了一些检查分析索引的技巧，以及如何避免慢速查询，而且也已经掌握了许多可以日后持续学习的知识。

查询优化一直是与应用系统相关的，每个都不一样，但是我们希望本章介绍的知识和技巧可以帮助大家来优化查询的性能。经验与方法还是非常有用的。养成监控和分析调优查询的良好习惯。

在这个过程中，你可以持续学习查询优化器的冷门知识，确保应用程序查询的效率。

当开发自己的应用时，这里有一些建议请大家要记住：

- 索引非常有用，但是也有成本——它们会让写入变慢。
- MongoDB通常在一个查询里使用一个索引，所以多个字段的查询需要复合索引才能更加高效。
- 当声明复合索引时顺序非常重要。
- 应该避免昂贵的查询。使用MongoDB的`explain`命令，它的昂贵查询日志以及分析器可以用来查找需要优化的查询。
- MongoDB提供几个构建索引的命令，但是这些通常包含开销，而且可能影响到应用程序。这意味着在流量和数据暴增之前应该尽可能早地优化查询和创建索引。
- 通过减少扫描的文档数量来优化查询。`explain`命令对于分析查询工作非常有用，它可以作为优化的向导。

建立索引和查询优化都会带来复杂性，早期的实验经验可能是最好的老师，给我们提供有价值的参考资料。

本章内容

- 为什么文本搜索如此重要
- 文本搜索基础
- 定义MongoDB文本搜索索引
- 使用MongoDB find()
- 使用带聚合统计的MongoDB文本搜索
- 使用不同语言的文本搜索

在第5章和6章里，介绍了如何构建查询和使用聚合函数，学习了如何使用复杂的查询语言来执行数据库查询。对于许多应用程序来说，使用这些查询已经足够了，但是有时候还需要处理非结构化的查询，或者尝试支持用户从巨大的类别里查找可能关联的商品，这时这种搜索可能不足。网站浏览者已经适应了使用Google或者Amazon来搜索期望的内容，而且更加依赖复杂的搜索技术。

本章里，我们会介绍MongoDB如何提供更加复杂的文本搜索引擎——比目前为止我们看到的查询都要复杂得多。这些额外功能包括快速单词搜索的索引、匹配精确字段、使用特定单词或者句子排除文档、支持多语言、基于匹配度对查询结果打分。虽然MongoDB文本搜索并非为了替换专业的搜索引擎，但是它也提供了足够强大的类似功能来满足某些需求，而不需要使用专门的搜索引擎。

我们来看看专业的搜索引擎提供了哪些搜索功能。在9.1.3小节里，我们会看到MongoDB提供的这些功能的子集。

如果你懂了，那为什么不使用呢？

在LinkedIn MongoDB的讨论组中，有些人问到MongoDB文本搜索和专业搜索引擎Elasticsearch的区别。MongoDB产品总监Kelly Stirman的回答如下：通常，Elasticsearch比MongoDB提供了更加丰富的功能。这很正常——它是专业的搜索引擎。而MongoDB文本搜

索引适合基本的搜索需求。如果已经在MongoDB里存储数据,那么文本索引会带来额外的部署开销,但是通常比同时部署MongoDB和Elasticsearch简单得多。

注意:我们可以在Radu Gheorghe的《Elasticsearch in Action》(Manning Publications, 2015)中获取更多Elasticsearch的信息。我们也可以阅读另外一本关于专业搜索引擎的书籍,也是基于Apache Lucene构建的:Trey Grainger 和 Timothy Potter编写的书《Solr in Action》(Manning Publications, 2014)。

9.1 文本搜索—不仅仅是模式匹配

Text searches—not just pattern matching

为满足日常生活需要,我们可能每天都要使用搜索引擎,次数可能不确定。

作为程序员,可能我们会使用搜索引擎来查询资料,帮助解决Bug。也可能晚上回家搜索Amazon或者其他电商网站,也可能在Manning.com出版社官方网站使用Google支持的自定义搜索来查询书籍。^[1]

如果访问Manning.com,就会看到网站右上角的搜索框的“Search manning.com”文字。输入关键字,比如Java,点击Search按钮,就会看到如图9.1所示的结果。

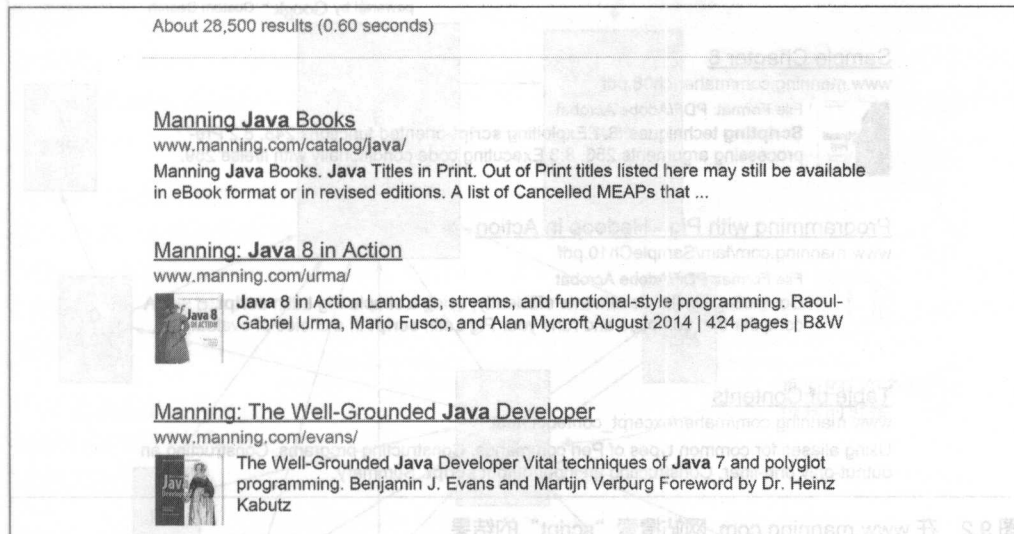


图 9.1 在 www.manning.com 网站搜索 java 关键字的结果

注意,因为搜索结果基于实时数据,所以结果可能不太一样。或许编写本书时刚出版的《Java 8 in Action》会被《Java 9 in Action》甚至更高版本Java图书取代。

^[1]【译者注】中国用户可能使用淘宝或者百度搜索商品。程序员一般使用国外的搜索引擎 Google 或者 Bing.com。

这个搜索的目的是给大家演示一下文本搜索功能的几个重要特点：

- 搜索时大小写不敏感（case-insensitive search），意味着无论使用“jAVA”还是“Java”或者“java”，我们看到的结果都是“Java”大小写共存的。
- 看不到任何包含JavaScript的搜索结果，即使有JavaScript图书名字包含了Java字符串。这是因为搜索引擎认为“Java”和“JavaScript”是不同的单词。

正如我们知道的，也可以使用正则表达式在MongoDB进行查询，只要指定模式匹配的规则即可。但是在MongoDB里，这种模式匹配搜索在大型的集合上可能很慢，由于无法利用索引，因此文本搜索引擎要做大量的筛选工作。复杂的MongoDB搜索都无法提供真正意义上的文本搜索功能。

我们来使用另外一个例子演示一下。

9.1.1 文本搜索与模式匹配


现在在Manning.com网站尝试一下第二个搜索。这次使用搜索词语“script”。我们会看到如图9.2所示的结果。

About 5,010 results (0.32 seconds)

powered by Google™ Custom Search

Sample Chapter 8

www.manning.com/maher/ch08.pdf




File Format: PDF/Adobe Acrobat

Scripting techniques. 8.1 Exploiting script-oriented functions 248. 8.2 Pre-processing arguments 256. 8.3 Executing code conditionally with if/else 259.

Programming with Pig - Hadoop in Action

www.manning.com/lam/SampleCh10.pdf



File Format: PDF/Adobe Acrobat

Computing similar documents efficiently, using a simple Pig Latin **script**. □ ... 2 A compiler that compiles and runs your Pig Latin **script** in a choice of evaluation ...

Table of Contents

www.manning.com/maher/excerpt_contents.html

Using aliases for common types of Perl commands. Constructing programs. Constructing an output-only one-liner, Constructing an input/output **script**. Summary.

图 9.2 在 www.manning.com 网站搜索“script”的结果

注意，这次搜索包含了图书名字包含“scripting”和“script”的结果，但是不包含“JavaScript”。这是因为搜索引擎执行的功能称为词根检索（stemming）机制，大家输入的单子被转换为词根或者“scripting”单词的起源词根——这个例子中就是“script”。搜索引擎必须明白存储和搜索的结果语言，知道“script”可以推测出“scripts”、“scripted”、“scripting”，而

不是“JavaScript”。

Web页面搜索使用了许多类似文本搜索的功能，而且它们也提供了其他的搜索功能。我们来看看这些功能，以及它们如何帮助用户解决问题。

9.1.2 文本搜索与网页搜索

Web页面搜索引擎包含与专业搜索引擎类似的功能，通常更多。Web页面搜索关注于页面搜索，这对于搜索万维网是个优势，但是对于搜索某个类别的商品可能是个劣势。这种搜索基于文档之间的关系，在专业的搜索引擎中是没有的，而MongoDB提供了这种关联支持，还有新的文本搜索功能。

Google最早使用的搜索引擎网页排名算法称为Page Rank。这个命名并不是因为它可以排名网页，而是因为它由Google的两位创始人之一Larry Page的名字命名。Page Rank根据页面超链接的重要性来标记页面的重要性或者权重。图9.3所示为Wikipedia维基百科的Page Rank分析。

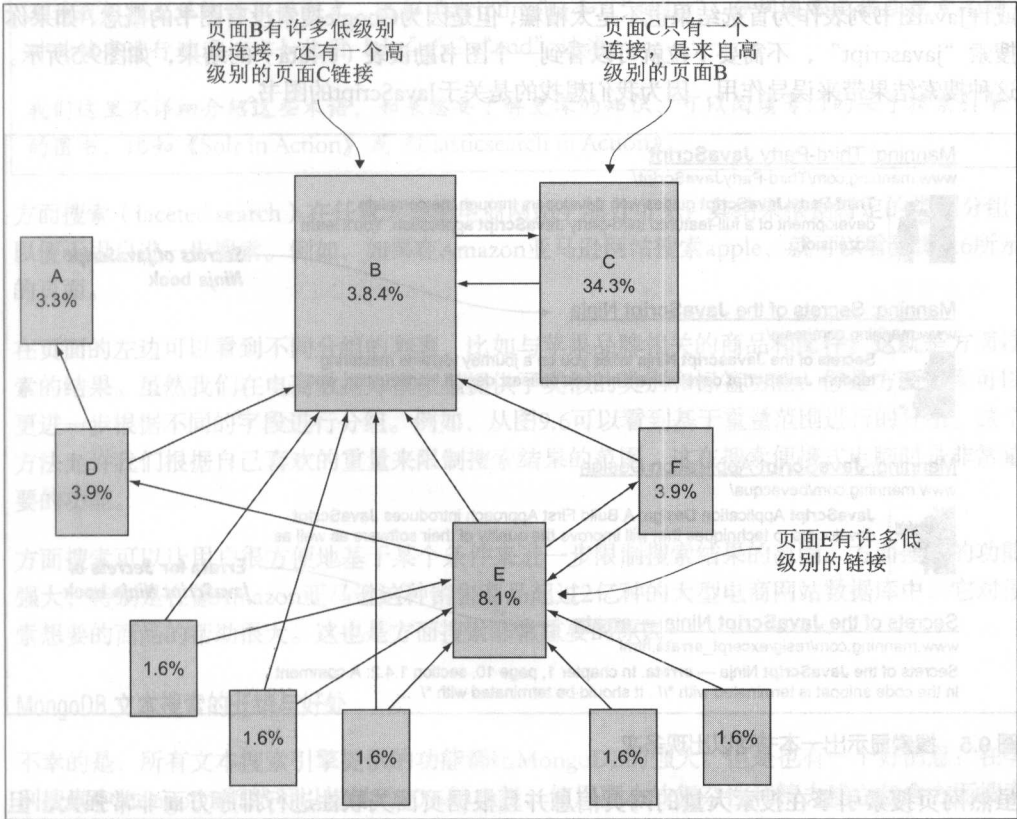


图 9.3 基于页面链接重要性的 PR 值

如图9.3所示，页面C与页面B几乎一样重要，因为它被一个非常重要的页面指向了：页面B。这个算法在大学数据挖掘的课程里已教授，也可以用来计算页面外联的超链接数量。此时，不仅页面B很重要，而且它还有一个外联页面，那个页面也很重要。注意，页面E也有很多超链接，但是这些超链接来自排名比较低的页面，所以页面E的评价不高。

Google现在使用了许多算法来评价页面重要性，据统计超过200种算法，已成为一个完整功能的搜索引擎。但是记住，页面搜索与我们要搜索某个类别的商品完全不同。

页面搜索会访问由数据库生成的页面，而不是数据库本身。例如，搜索java的页面结果如图9.4所示。第一个搜索结果不是商品，而是Manning出版社关于java的图书列表链接页面。

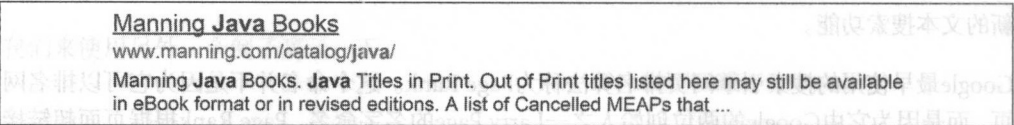


图 9.4 搜索结果不仅仅是图书

或许Java图书列表作为首选结果也不是太糟糕，但是因为Google搜索没有图书的概念，如果你搜索“javascript”，不需要下拉就可以看到一個图书勘误表（errata）的结果，如图9.5所示。这种搜索结果带来误导作用，因为我们想找的是关于JavaScript的图书。

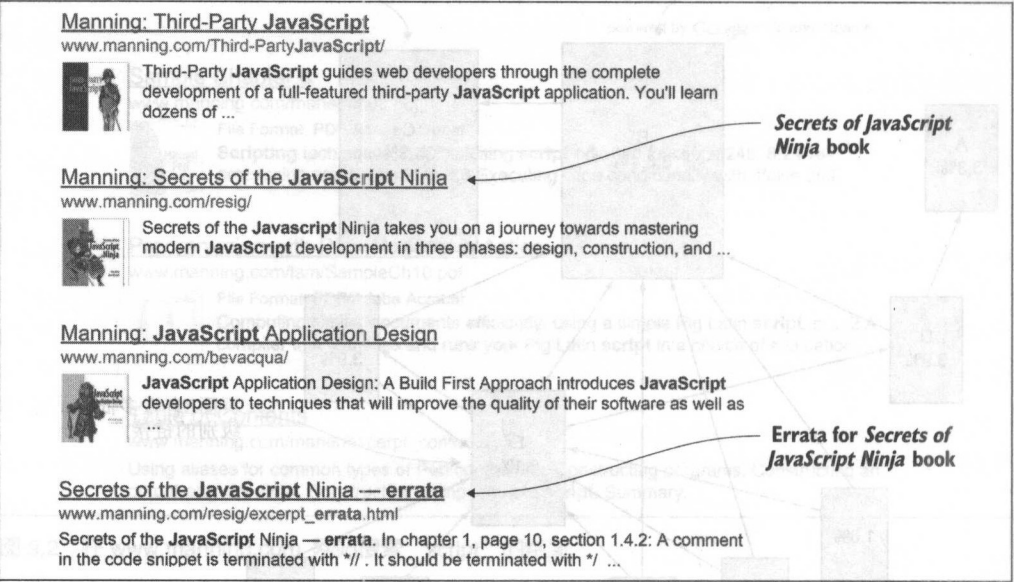


图 9.5 搜索显示出一本书可以出现多次

虽然网页搜索引擎在搜索大量的网页信息并且根据页面关联性进行排序方面非常强大，但是对于搜索某个商品数据库类型的问题确实不是强项。要解决这种问题，我们可以研究完整功能的文本搜索引擎，它们可以搜索单一的商品数据库，比如，我们要在Amazon上进行

查找商品。

9.1.3 MongoDB 文本搜索与专业搜索引擎

专业的搜索引擎不仅可以为网页建立索引，还可以为大型数据库建立索引。文本搜索引擎可以提供比如拼写错误纠正、查找提示，以及关联标准——许多Web搜索引擎都可以做到的这些功能。但是专业的搜索提供更多的改进，比如方面（facets）、同义词库、词根提取算法以及自定义分词词典。

方面（facets）、同义词库、词根提取算法以及自定义分词词典？

如果你没有专门研究过搜索引擎，可能会好奇，这些词汇的含义是什么？简单来说，方面（facets）允许我们根据某个特性来分组商品，比如图9.6所示中左侧展示的“Laptop Computers”类别。同义词库允许我们使用不同的词表示相同的含义。例如，搜索“intelligent”时，可能也想知道搜索“bright”和“smart”的结果。正如9.1.1节介绍的，词根分析（stemming）允许我们找出单词的不同形式，比如“scripting”和“script”。分词通常是指在搜索之前对关键字进行过滤，比如使用“the”“a”“and”过滤。

我们这里不详细介绍这些术语，如果想要了解更深的知识，可以阅读专门的关于搜索引擎的图书，比如《Solr in Action》或《Elasticsearch in Action》。

方面搜索（faceted search）在任意大型的电商网站上随处可见，其结果根据特定的类别分组，以便于用户进一步搜索。例如，如果在Amazon亚马逊网站搜索apple，就可以看到图9.6所示的页面。

在页面的左边可以看到不同分组的列表，比如与苹果品牌相关的商品和配件。这就是方面搜索的结果。虽然我们在电商数据库模型里提供了类似的类别和标签功能，但是方面搜索可以更进一步根据不同的字段进行分组。例如，从图9.6可以看到基于重量范围进行的分组。这个方法允许我们根据自己喜欢的重量来限制搜索结果的范围，这在搜索便携式电脑时是非常重要的功能。

方面搜索可以让用户很方便地基于某个条件来进一步限制搜索结果的范围。方面搜索的功能强大，特别是在像Amazon亚马逊这种销售商品超过2亿种的大型电商网站数据库中，它对搜索想要的商品的帮助很大。这也是方面搜索非常重要的原因。

MongoDB 文本搜索的开销与好处

不幸的是，所有文本搜索引擎提供的功能都比MongoDB的强大。但是也有一个好消息：在类别搜索中MongoDB仍然可以提供约80%的功能，使用更小的复杂性和精力建立包含方面搜索和词汇提示的全文搜索引擎。那么MongoDB能做什么呢？



图 9.6 在 amazon 网站使用“apple”进行搜索

- 基于词根分析的自动化实时索引
- 根据字段名可选分配权重
- 多语言支持
- 分词
- 精确词汇匹配
- 根据给定短语或单词排除某些结果的能力

注意：与全文搜索引擎不同，MongoDB不允许我们编辑分词列表。这里还有一个用户请求，要求增加分词列表：<https://jira.mongodb.org/browse/SERVER-10062>。

这些功能在为价格定义索引的时候可以用，可以让我们使用文本搜索功能而不需要把数据库全部拷贝到整个搜索引擎中。这个方法也避免了专业搜索引擎的额外管理开销。如果MongoDB提供了足够的功能，那么也是个不错的权衡。

现在来看看MongoDB提供的支持细节（非常简单）：

- 首先，定义文本搜索需要的索引。
- 然后，在基本查询和聚合框架里使用文本搜索。

需要的关键组件是MongoDB 2.6及以后的版本。MongoDB 2.4中的文本搜索还处于实验阶段，

到MongoDB 2.6该功能才成为默认的功能，而且文本关联搜索功能与find()和aggregate()函数集成。

你应该知道的 MongoDB 的文本搜索功能

第8章可以帮助我们理解索引，文本搜索索引应该非常简单。如果为基本查询或者聚合框架使用文本搜索，就必须熟悉第5章的相关内容，包含执行基本的查询。第6章包含了如何使用聚合框架。

MongoDB 文本搜索:一个简单例子

在深入MongoDB文本搜索底层原理之前，我们来看一下电商网站的例子。首先，我们需要定义索引，从指定字段开始。我们将会在9.3节里详细介绍如何使用文本索引，但是这里有个使用电商products集合的简单例子：

```
db.products.createIndex(  
  {name: 'text',  
   description: 'text',  
   tags: 'text'}  
);
```

索引名字段
索引描述字段
索引标签字段

索引指定了products集合中的三个字段：name,description,tags。现在我们来看一下在products集合里搜索gardens的简单例子：

```
> db.products  
  .find({$text: {$search: 'gardens'}},  
        {_id:0, name:1,description:1,tags:1})  
  .pretty()  
{  
  "name" : "Rubberized Work Glove, Black",  
  "description" : "Black Rubberized Work Gloves...",  
  "tags" : [  
    "gardening"  
  ]  
}  
{  
  "name" : "Extra Large Wheel Barrow",  
  "description" : "Heavy duty wheel barrow...",  
  "tags" : [  
    "tools",  
    "gardening",  
    "soil"  
  ]  
}
```

搜索文本“gardens”字段
“gardening”匹配搜索
“gardening”匹配搜索

这个例子演示了MongoDB文本搜索的关键方面，以及它与普通文本搜索的不同之处。在这个例子里，搜索“gardens”会导致对词根“garden”的搜索。可以找到带有gardening标签的2个商品，它们已经被词根分析处理，并且在garden下建立了索引。

下面几节里，我们要学习更多关于MongoDB文本搜索工作原理的内容。首先，我们来下载一个大数据集的例子数据库。

9.2 下载曼宁图书类别数据

Manning book catalog data download

对于电商网站的例子，本书目前为止介绍的知识点已经足够了。本章我们介绍了一个大数据集用来演示MongoDB文本搜索功能的优点和缺点。这个数据集包含曼宁出版社的图书类别，仅限于本书编写时的类别数据。如果要运行例子，可以通过下面的步骤来下载数据到本地MongoDB数据库：

- 在本书包含的代码里找到chapter9文件夹，把catalog.books.json复制到本地计算机方便的位置。
- 运行这里显示的命令。可能要修改文件的前缀，catalog.books.json，包含保存文件的路径参数。

```
mongoimport --db catalog --collection books --type json --drop
--file catalog.books.json
```

应该可以看到与列表9.1显示的数据相似的结果。请注意findOne()函数返回随机选择的文档数据。

列表9.1 加载数据到books集合里。

```
> use catalog                                ← 切换到类别数据库
switched to db catalog
> db.books.findOne()                         ← 显示类别中随机选择的书
{
  "_id" : 1,
  "title" : "Unlocking Android",
  "isbn" : "1933988673",
  "pageCount" : 416,
  "publishedDate" : ISODate("2009-04-01T07:00:00Z"),
  "thumbnailUrl" : "https://s3.amazonaws.com/AKIAJC5RLADLUMVRPFDQ
.book-thumb-images/ableson.jpg",
  "shortDescription" : "Unlocking Android: A Developer's Guide
provides concise, hands-on instruction for the Android operating system and
development tools. This book teaches important architectural concepts in a
straightforward writing style and builds on this with practical and useful
examples throughout.",
  "longDescription" : "Android is an open source mobile phone
platform based on the Linux operating system and developed by the Open
Handset Alliance, a consortium of over 30 hardware, software and telecom
..."
  "status" : "PUBLISH",
  "authors" : [
    "W. Frank Ableson",
    "Charlie Collins",
```

```
"Robi Sen"
],
"categories" : [
  "Open Source",
  "Mobile"
]
```

列表也显示了文档的数据结构。每个文档都包含下面的字段：

- title——图书标题。
- isbn——国际标准图书编号（ISBN）。
- pageCount——页面数量。
- publishedDate——发行日期（只有当status字段为PUBLISH时才有）。
- thumbnailUrl——封面缩略图URL地址。
- shortDescription——内容简介。
- longDescription——详细介绍。
- status——图书状态，PUBLISH或者MEAP。
- authors——作者名字数组。
- categories——图书类别数组。

现在我们已经加载了图书列表，接下来就为它创建文本索引。

9.3 定义文本搜索索引

Defining text search indexes

文本索引与7.2.2节看到的索引类似，包含创建索引和删除索引。一个重要的不同是一个集合只有一个文本索引。

下面是books集合中文本索引定义的代码例子：

```
db.books.createIndex(
  {title: 'text',
   shortDescription: 'text',
   longDescription: 'text',
   authors: 'text',
   categories: 'text'},
  {weights:
    {title: 10,
     shortDescription: 1,
     longDescription: 1,
     authors: 1,
     categories: 5}}
```

指定文本索引字段

权重设置

下几节里，我们会看到更多关于 MongoDB 文本搜索的“内幕”。首先，我们来下载一个大数据集的例子数据库。

7.2.2 小节中普通索引和文本索引还有一些重要的区别：

- 索引字段后指定 `text`，而不是 `1` 或 `-1`。可以为文本索引指定任意多个字段，而且所有的字段都会像单个字段一样被搜索。每个集合只有一个文本索引，但是它可以为任意多个字段建立索引。

不用担心字段的权重。权重允许我们为搜索结果里的字段指定重要性。我们将会在 9.4.2 节中深入讲解字段权重。

9.3.1 文本索引的大小

每个索引入口为文档中的每个词根单词创建。正如你可能想像的，文本索引很大。为了减少入口数量，某个单词（又叫做分词）被忽略。

我们在讨论方面搜索时提到过，分词是通常不会被搜索的单词。在英语中分词包括“the”“an”“a”“and”等。试图搜索这种单词没有多大意义，因为它几乎会返回集合中的所有文档数据。

列表 9.2 展示了 `stats()` 命令在 `books` 集合上的执行结果。`stats()` 命令展示了 `books` 集合的大小，还有集合中索引的大小。

列表 9.2 集合统计显示了索引的名字和使用空间。

```
> db.books.stats()
{
  "ns" : "catalog.books",
  "count" : 431,
  "size" : 772368,
  "avgObjSize" : 1792,
  "storageSize" : 2793472,
  "numExtents" : 5,
  "nindexes" : 2,
  "lastExtentSize" : 2097152,
  "paddingFactor" : 1,
  "systemFlags" : 0,
  "userFlags" : 1,
  "totalIndexSize" : 858480,
  "indexSizes" : {
    "_id_" : 24528,
    "title_text_shortDescription_text_longDescription_text_authors_text_
categories_text" : 833952
  },
  "ok" : 1
}
```

books 集合的大小

文本搜索索引的名字和使用空间

注意：`books` 集合的大小（列表 9.2 中的 `size`）是 772 368。从列表中的 `indexSizes` 字段可以看到文本搜索索引的名字和大小。文本索引的大小是 833 952——比 `books` 集合还大！第一眼

看到可能会吓一跳，但是要记住，索引必须包含集合文档中每个唯一的词根单词。而且还有指向被建立索引的文档。虽然删除了分词单词，但必须重复绝大部分被建立索引的文本，并为每个单词添加指向最初文档的指针。

另外一个要重点注意的问题是索引的名字：

```
"title_text_shortDescription_text_longDescription_text_authors_text_categories_text."
```

MongoDB命令空间的最大长度是123字节。如果索引的文本字段比较多，就很可能超过123个字节的限制。我们来看一下如何通过定义索引名称来避免超出限制的问题。我们也会展示一个简单的指定集合索引名称的方法。

9.3.2 分配索引名字并为集合里的所有字段建立索引

在MongoDB里，命名空间是数据库和集合名字的连接体，中间有个圆点。命名空间最多有123个字节。在前面的例子里，索引的命名空间名字的长度是84个字符。

有许多种方法可以避免这种问题。首先，对于所有的MongoDB索引，都有指定索引名字的参数选项，如下所示：

```
db.books.createIndex(  
  {title: 'text',  
   shortDescription: 'text',  
   longDescription: 'text',  
   authors: 'text',  
   categories: 'text'},  
  
  {weights:   
    {title: 10,   
     categories: 5},  
  
  name : 'books_text_index'  
});
```

使用 1 以外的值指定权重

用户定义的索引名字

这个例子还指定了title和categories的权重，但是其他字段的权重默认值是1。我们会在第9.4.3小节里介绍更多的关于排名积分影响搜索结果排序的问题。

注意：如果某个索引已经存在，即使使用不同的名字也不能再次创建它（错误信息：所有的索引已经存在）。在这种情况下，首先需要使用dropIndex()删除已有的索引，然后使用新名字重新创建。

通配符字段名称

文本搜索索引也有个专门的通配符字段名称: \$**。这个名字表示要匹配任意字段中包含的字符串。对于使用通配符的文本索引，默认的索引名字是\$**_text，因此，可以避免123个字


```
{ "_id" : 17, "title" : "MongoDB in Action" }
:
```

对于这个查询，搜索字符串被分割为多个单词，分词被删除，剩余的单词提取词根，然后MongoDB使用文本索引执行搜索。这个过程如图9.7所示。

图中只有一个分词“in”，提取词根后的结果与原来的单词一样。MongoDB接下来使用这些单词搜索，不区分大小写，一共两次使用索引：一次是针对“mongodb”，一次是针对“action”。结果可能是包含两个单词之一的任意文档，等价于or（或）条件搜索。

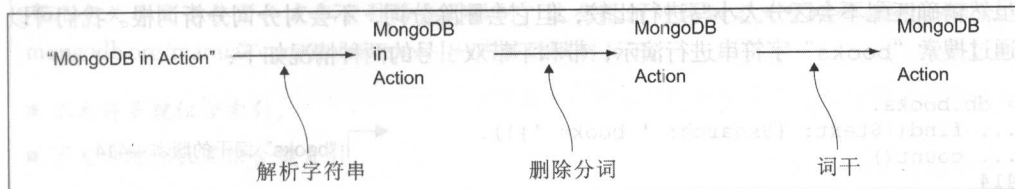


图 9.7 文本搜索字符串处理

既然已经学习了简单的文本查询，那么现在就来继续学习一些高级的搜索知识。

9.4.1 更复杂的搜索

除了搜索任意数量的单词，指定“or”来组合多个搜索条件，MongoDB还允许做下面的事情：

- 使用“and”代替“or”表示并且搜索关系。
- 执行精确短语匹配。
- 使用特定单词排除文档。
- 使用特定短语排除文档。

下面来看一下如何让结果文档包含某个给定的单词。

我们已经看了对于“mongodb in action”短语的搜索，不仅返回关于MongoDB的图书，还包括任意包含action单词的图书。如果使用双引号包含搜索单词，它的意思是单词必须都包括在结果文档里。下面是一个例子：

```
db.books.find({$text: {$search: ' "mongodb" in action'}}) ← 双引号中的“mongodb”必须出现
```

此查询只返回图书标题包含单词“mongodb”的查询：

```
{ "title" : "MongoDB in Action" }
{ "title" : "MongoDB in Action, Second Edition" }
```

精确匹配短语

短语也可以使用双引号，所以如果指定短语second edition，只显示了本书的第2版，因为多个短语使用了“and”搜索：

```
> db.books.  
... find({$text: {$search: ' "mongodb" "second edition" '}}),  
... { _id:0, title:1})  
{ "title" : "MongoDB in Action, Second Edition" }
```

“second edition”也是必须的

虽然精确匹配不会区分大小写进行比较，但它会删除分词，不会对分词分析词根。我们可以通过搜索“books”字符串进行演示，带和不带双引号的两种情况如下：

```
> db.books.  
... find({$text: {$search: ' books '}}).  
... count()  
414  
  
> db.books.  
... find({$text: {$search: ' "books" '}}).  
... count()  
21
```

“books” 词干的版本—414

精确词“books”—21

这里看到，当我们不使用双引号指定“books”时，MongoDB提取词根后可以搜索到414个结果。当指定精确匹配，使用双引号括住“books”，MongoDB只会返回精确包含此单词的文档图书，总共21个。你实际测试的结果可能不同，因为数据库数据集合可能不一样。

使用专门的单词或者短语排除文档

要使用单词排除文档，可以在该单词前面加上“-”号即可。例如，如果想搜索带有单词的“MongoDB”图书，但是又想排除包含“second”的图书，那么可以这样做：

```
> db.books.  
... find({$text: {$search: ' mongodb -second '}}),  
... { _id:0, title:1 })  
{ "title" : "MongoDB in Action" }
```

使用“second.”排除单词

注意：第2行和第3行的前三个圆点是mongo shell自动添加的，这是对多于一行的输入命令自动补齐的符号——这个例子代码有3行。

与此类似，我们可以使用双引号包含某个短语，并且加上“-”号以便排除文档：

```
> db.books.  
... find({$text: {$search: ' mongodb -"second edition" '}}),  
... { _id:0, title:1})  
{ "title" : "MongoDB in Action" }
```

使用“second edition.”排除文档

更复杂的搜索条件

我们可以使用find()和文本搜索结合起来执行更多条件的查询。例如，如果要搜索所有的状态

为“MEAP”的Java图书，可以使用：

```
> db.books.  
... find({$text: {$search: 'mongodb'}, status: 'MEAP'},  
... {_id:0, title:1, status:1})  
{ "title" : "MongoDB in Action, Second Edition",  
"status" : "MEAP"}
```

状态必须是 MEAP

组合文本搜索条件的限制

有几个关于文本搜索和文本索引结合的限制。这些限制的详细定义可以参考<http://docs.mongodb.org/manual/core/index-text/>。一些键的限制如下：

- 不允许多键组合索引。
- 不允许地理位置组合键索引。
- 如果查询表达式包含\$text，无法使用hint()。
- 排序操作无法从文本索引获取排序顺序。

如果添加了更多的分词，比如“the”，修改搜索词语不使用词根分析，就可以看到相同的结果。这个不能证明分词被忽略，或者无词根分析的单词与词根分析的单词一样处理。

要证明它，我们来看一下文本积分搜索，确认我们接收同样的分数，即使额外的分词或者不同的单词使用相同的词根。我们来看一下什么是文本搜索积分，以及如何在结果中使用它。

9.4.2 文本搜索分数

文本搜索分数提供了一个数字，表示文档的相关性，这个数值依据关键字在文档中出现的次数而定。评分也使用了创建索引时分配给不同字段的权重，9.3节里做过介绍。

要显示文本分数，可以在find()里使用score: { \$meta:"textScore" }这种投影字段。注意，分配文本分数的名字——这里是score——其实可以任意指定喜欢的名字。列表9.3展示了早期一样的例子，但是显示了搜索分数，几乎使用了相同的搜索字符串。请注意，输出结果与之前的例子有些不同。

列表9.3 显示文本搜索分数

```
> db.books.  
... find({$text: {$search: 'mongodb in action'}}),  
... {_id:0, title:1, score: { $meta: "textScore" }}).  
... limit(4);  
{ "title" : "Machine Learning in Action", "score" : 16.83933933933934 }  
{ "title" : "Distributed Agile in Action", "score" : 19.371088861076345 }  
{ "title" : "PostGIS in Action", "score" : 17.67825896762905 }  
{ "title" : "MongoDB in Action", "score" : 49.48653394500073 }
```

搜索 “mongodb in action”

结果中包含文
本搜索分数

第一个搜索
字符串的文
本搜索分数


```
> db.books.
...   find({$text: {$search: 'the mongodb and actions in it'}}),
...   { _id:0, title:1, score: { $meta: "textScore" }}).
... limit(4);
{ "title" : "Machine Learning in Action", "score" : 16.83933933933934 }
{ "title" : "Distributed Agile in Action", "score" : 19.371088861076345 }
{ "title" : "PostGIS in Action", "score" : 17.67825896762905 }
{ "title" : "MongoDB in Action", "score" : 49.48653394500073 }
```

第二个文本分词等价于第一个分数集合

带有额外分词的第二个文本字符串和复数单词“actions”

在这个列表代码中，搜索字符串从“mongodb in action”变成了“the mongodb and actions in it”。使用了action的复数形式，而且增加了多个分词。正如我们看到的，文本分数在两种情况下都是一样的，证明分词单词实际情况下被忽略，而其他单词进行了词根分析。

分配权重影响单词重要性

在9.3.2小节创建的索引里，我们注意到一个字段叫weights。这个权重字段影响某个字段被找到的重要性。默认的字段权重是1，但是正如我们看到的，我们为categories字段分配了权重5，而为title分配了权重10。这意味着categories字段被发现的概率是其他字段的5倍，title字段的是其他字段的10倍，是categories字段的2倍。这都会影响到分配给文档的分数：

```
db.books.createIndex(
  {'$**': 'text'},
  {weights:
    {title: 10,
     categories: 5}
  });
```

为字段指定 1 以外的权重

这个搜索在想要查询所有包含mongodb或action字段的图书时非常有用。但是对于绝大多数搜索，可能先看最相关的结果。我们来看看如何实现。

9.4.3 根据文本搜索分数排序结果

根据相关性对结果进行排序，其实现方式与上一个例子的一样。事实上，要根据文本搜索分数排序，就必须在find()的投影规定里包含\$meta函数。这里是个例子代码：

```
db.books.
  find({$text: {$search: 'mongodb in action'}},
    {title:1, score: { $meta: "textScore" }}).
  sort({ score: { $meta: "textScore" } })
```

文本分数投影

根据文本积分排序

这是根据文本分数排序的文本搜索结果列表：

```
{ "_id" : 17, "title" : "MongoDB in Action", "score" : 49.48653394500073 }
{ "_id" : 186, "title" : "Hadoop in Action", "score" : 24.99910329985653 }
{ "_id" : 560, "title" : "HTML5 in Action", "score" : 23.02156177156177 }
```

正如之前提到的，我们可以为文本搜索分数起任意名字。在这里叫score，但是也可以使用textSearchScore这种名字。但是，sort()函数里指定的名字必须与前面find()函数里指定的名字一样。另外，不能使用文本排序字段指定排序方式（升序或者降序）。这种排序结果通常是从最高到最低排序，通常比较合理，因为我们通常需要关联最紧密的结果。如果某些原因需要最不相关的结果在最前面，就可以使用下一节里介绍的文本搜索聚合框架。既然已经看了如何在find()里使用文本搜索，现在就来看看怎么使用聚合框架。

投影字段\$meta:"textScore"

正如我们在5.1.2小节里学习的，可以使用投影来限制find()函数返回的字段。如果在查找投影里指定了任意字段，那么只会返回这些指定的字段。

如果在投影里包含了文本搜索的分数，文本搜索只会根据分数排序。这是不是意味着如果要根据文本分数进行排序，就必须指定其他要返回的字段？

幸运的是，不需要。如果在投影里只指定文本分数，所有其他的文本字段会自动包含进来，还有文本的分数字段。

9.5 聚合框架文本搜索

Aggregation framework text search

正如大家在第6章里学习的一样，使用聚合框架，可以把多个文档中的数据组个成一个新的文档，从而获取单一文档无法提供的数据。在本节里，我们会学习如何在聚合框架里使用文本搜索功能。后面我们会看到聚合框架提供了比find()还多的文本搜索功能。

在9.4.3小节中，我们看了一个简单的例子，使用mongodb in action搜索图书，然后根据文本分数排序：

```
db.books.find({$text: {$search: 'mongodb in action'}}).
  {title:1, score: { $meta: "textScore" }}).
  sort({ score: { $meta: "textScore" } })
```

搜索带有 mongodb 或者 action 单词的文档

文本分数投影

根据文本积分排序

使用聚合框架，我们可以得到相同的结果：

```
db.books.aggregate([
  { $match: { $text: { $search: 'mongodb in action' } } },
  { $sort: { score: { $meta: 'textScore' } } },
])
```

搜索带有 mongodb 或者 action 单词的文档

文本分数投影

```

    { $project: { title: 1, score: { $meta: 'textScore' } } } } ← "title"
  )
 根据文本分数排序

```

正如期望的，代码会生成与之前的例子一样的结果：

```

{ "_id" : 17, "title" : "MongoDB in Action", "score" : 49.48653394500073 }
{ "_id" : 186, "title" : "Hadoop in Action", "score" : 24.99910329985653 }
{ "_id" : 560, "title" : "HTML5 in Action", "score" : 23.02156177156177 }
{ "_id" : 197, "title" : "Erlang and OTP in Action", "score" :
22.069632021922096 }

```

注意：两个版本的文本搜索都使用了相同的构造函数来指定查找和匹配条件、投影字段以及排序条件。但是我们承诺过，聚合框架可以做得更多。例如，可以通过包装\$sort和\$project操作符、简化\$sort操作符，重写之前的例子：

```

db.books.aggregate(
  [
    { $match: { $text: { $search: 'mongodb in action' } } },
    { $project: { title: 1, score: { $meta: 'textScore' } } }, ← 根据文本分数降序排序
    { $sort: { score: -1 } }
  ]
)

```

第二个聚合例子的最大不同是，与使用find()函数不一样，我们选择可以引用前面\$project操作里的score字段。注意，我们采用降序排列结果，因此使用score: -1代替score: 1。但是它也提供了选项，可以score: 1来指定先显示最低关联分数的图书。

聚合框架使用\$text搜索，有一些限制：

- 使用\$text函数的\$match操作符必须是管道里的第一个操作，而且必须跟着其他的\$meta:'textScore'操作符。
- \$text函数只能在管道里出现一次。
- \$text函数不能与\$or或者\$not一起使用。

使用\$match文本搜索字符串，格式与find()命令的格式一样：

- 如果单词或短语放在双引号里，文本必须精确匹配。
- 如果单词或短语前缀用了负号(-)，就是使用此条件排除文档。

在下一节里，我们会学习到如何使用此功能来访问文本分数进行更多的自定义查询。

如果你看过前面例子使用“mongodb in action”字符串搜索的代码，就可能好奇，为什么不包含第2版和第1版的结果？要找出原因，使用相同的字符串，但是双引号包含monogdb，这样会发现只会返回包含mongodb单词的文档：

```

> db.books.aggregate(
... [
...   { $match: { $text: { $search: ' "mongodb" in action ' } } },

```

```
... { $project: { _id:0, title: 1, score: { $meta: 'textScore' } } }
... }
... )
{ "title" : "MongoDB in Action", "score" : 49.48653394500073 }
{ "title" : "MongoDB in Action, Second Edition", "score" : 12.5 }
```

当看到这些结果时，就应该明白了为什么使用monogdb in action无法搜索到结果，因为它没有满足精确匹配条件。但是现在的问题是，为什么第2版的分数这么低？如果只查找第2版，答案就更明显了：

```
> db.books.findOne({"title" : "MongoDB in Action, Second Edition"})
{
  "_id" : 755,
  "title" : "MongoDB in Action, Second Edition",
  "isbn" : "1617291609",
  "pageCount" : 0,
  "thumbnailUrl" :
"https://s3.amazonaws.com/AKIAJJC5RLADLUMVRPFDQ.book-thumbimages/
banker2.jpg",
  "status" : "MEAP",
  "authors" : [
    "Kyle Banker",
    "Peter Bakkum",
    "Tim Hawkins",
    "Shaun Verch",
    "Douglas Garrett"
  ],
  "categories" : [ ]
}
```

正如结果所示，数据是在图书出版之前输入的，第2版的信息没有shortDescription或longDescription字段。对于许多没有出版的书都有这样的问题，因此，这种书籍的匹配分数也比较低。

可以使用聚合框架的灵活性来补偿这个问题。一种方法是把分数乘以某个因子——比如3——如果某个文档不包含longDescription字段，则可以这样做。列表9.4展示了例子代码。

列表9.4 如果没有longDescription字段就添加文本乘法器。

```
> db.books.aggregate(
... [
... { $match: { $text: { $search: 'mongodb in action' } } },
... { $project: {
... title: 1,
... score: { $meta: 'textScore' },
... multiplier: { $cond: [ '$longDescription', 1.0, 3.0 ] } } },
... },
... { $project: {
... _id:0, title: 1, score: 1, multiplier: 1,
... adjScore: { $multiply: [ '$score', '$multiplier' ] } } },
... ],
... )
```

计算乘法器：如果
longDescription 不存在，则倍数为 3

计算乘法器：
分数×倍数

```

...
...      { $sort: { adjScore: -1 } }
...    ]
...  );
{ "title" : "MongoDB in Action", "score" : 49.48653394500073,
"multiplier" : 1, "adjScore" : 49.48653394500073 }
{ "title" : "MongoDB in Action, Second Edition", "score" : 12.5,
"multiplier" : 3, "adjScore" : 37.5 }
{ "title" : "Spring Batch in Action", "score" : 11.666666666666666,
"multiplier" : 3, "adjScore" : 35 }
{ "title" : "Hadoop in Action", "score" : 24.99910329985653,
"multiplier" : 1, "adjScore" : 24.99910329985653 }
{ "title" : "HTML5 in Action", "score" : 23.02156177156177,
"multiplier" : 1, "adjScore" : 23.02156177156177 }

```

根据调整后的
分数降序排序

第二版在列
表中的第二
位置

正如我们在管道的第一个\$project操作符里看到的，通过判断longDescription是否存在来决定是否乘以倍数因子。如果longDescription不存在，则为false，如果存在，则使用\$cond函数来设置1.0乘积因子。如果longDescription不存在，就设置3.0的乘积因子。

然后管道里的第2个操作符\$project，它可以把分数乘以1.0或3.0来调整分数。最后，按照调整后的分数降序排列结果。

正如你看到的，MongoDB的文本搜索有其局限性。缺少字段可能导致错过一些搜索结果。MongoDB文本搜索也提供了一些改进搜索，强制搜索结果包含某些关键字。聚合框架也提供了额外的灵活性和功能，而且在扩展搜索结果时非常有用。

现在既然已经学习了MongoDB基本和高级的搜索功能，现在可以来处理另外一个复杂的问题了：支持搜索非英语语言。

9.6 文本搜索语言

Text search languages

MongoDB文本搜索的强大功能依赖于词根处理机制。搜索action与搜索actions的结果一样，因为它们有相同的词根。但是词根处理是与语言相关的。

MongoDB无法识别复数或者其他没有词根处理的非英语语言，除非我们告诉MongoDB使用了哪种语言。

有三条途径可以告诉MongoDB我们使用了何种语言：

- 索引中——可以为集合指定默认的语言。
- 插入文档时——我们可以重写某个文档或者字段默认的语言，告诉MongoDB和索引指定的默认语言不同。
- 当find()或aggregate()执行文本搜索时——可以告诉MongoDB搜索使用的语言。

词根处理和分词：简单但有局限性

现在mongoDB使用了“simple language-specific suffix stemming” (see <http://docs.mongodb.org/manual/core/index-text/>)词根处理规范。各种不同的词根分析算法，包括后缀剥离的详细介绍展示在<http://en.wikipedia.org/wiki/Stemming>中。

如果使用的语言不支持此后缀剥离算法，比如中文，或者希望使用不同的或者自定义词根分析器，则最好的办法是找到一个功能更全的文本搜索引擎。

简单来说，MongoDB基于不同的语言使用了不同的分词词典，它不允许自定义分词词典。当然这是专业搜索引擎支持的功能。

9.6.1 在索引里指定语言

回到9.3.2小节创建的索引例子中，我们可以修改索引定义，加上一个默认的语言。在修改books集合默认语言之前，运行下面的搜索命令，应该是搜索不到结果，因为搜索的是分词“in”。记住，分词是不会建立索引的。

```
> db.books.find({$text: {$search: 'in '}}).count()
```

现在删除之前的索引，创建相同的索引，但是使用语言是法语：

```
db.books.dropIndex('books_text_index');
```

删除 books 上的文本索引

```
db.books.createIndex(
  {'$**': 'text'},
  {weights:
    {title: 10,
     categories: 5},
   name: 'books_text_index',
   default_language: 'french'
});
```

使用法语添加新索引

现在如果运行之前的find()，会看到一些图书，因为在法语里，“in”不是分词：

```
> db.books.find({$text: {$search: 'in '}}).count()
334
```

如果检查books集合上的索引，会看到现在的语言是法语：

```
> db.books.getIndexes()
[
  {
    "v": 1,
    "key": {
```



```

        "_id": 1,
        "name": "_id_",
        "ns": "catalog.books",
    },
    {
        "v": 1,
        "key": {
            "_fts": "text",
            "_ftsx": 1
        },
        "name": "books_text_index",
        "ns": "catalog.books",
        "weights": {
            "$**": 1,
            "categories": 5,
            "title": 10
        },
        "default_language": "french",
        "language_override": "language",
        "textIndexVersion": 2
    }
]

```

默认的索引文件是法语

9.6.2 在文档里指定语言

在插入指定语言的例子文档之前，修改索引语言回到英语上，运行下面的命令完成：

```
db.books.dropIndex('books_text_index');
```

```
db.books.createIndex(
    {'$**': 'text'},
```

```
    {weights:
      {title: 10,
       categories: 5},
```

```
    name: 'books_text_index',
```

```
    default_language: 'english'
  )
```

指定默认语言为英语

```
);
```

现在插入法语语言文档：

```

db.books.insert({
  _id: 999,
  title: 'Le Petite Prince',
  pageCount: 85,
  publishedDate: ISODate('1943-01-01T01:00:00Z'),
  shortDescription: "Le Petit Prince est une oeuvre de langue française,
la plus connue d'Antoine de Saint-Exupéry. Publié en 1943 à New York
simultanément en anglais et en français. C'est un conte poétique et
philosophique sous l'apparence d'un conte pour enfants.",
  status: 'PUBLISH',

```

```

    authors: ['Antoine de Saint-Exupéry'],
    language: 'french'
  })

```

指定语言为
“french”

在定义索引的时候，如果想要使用其他单词而不是language，MongoDB文本搜索也允许我们修改指定文档语言的字段名字。也可以指定文档的某些字段是不同的语言。我们可以阅读更多关于此主题的内容：[//docs.mongodb.org/manual/tutorial/specify-language-for-text-index/](https://docs.mongodb.org/manual/tutorial/specify-language-for-text-index/)。

现在我们插入了一个法语文档，看看如何搜索法语内容。

9.6.3 在搜索中指定语言

使用不同的语言搜索，结果差别很大。记住，语言可以影响MongoDB词根处理和定义分词的方式。我们来看看指定的语言如何影响索引和搜索。我们的第一个例子，即下面的代码列表9.5展示了词根处理对于文档索引和搜索分词的影响。

列表9.5 语言如何影响词根处理

```

> db.books.find({$text: {$search:
  'simultanment', $language: 'french'}}, {title:1})
{ "_id" : 999, "title" : "Le Petit Prince" }

> db.books.find({$text: {$search: 'simultanment'}}, {title:1})
{ "_id" : 186, "title" : "Hadoop in Action" }
{ "_id" : 293, "title" : "Making Sense of Java" }
{ "_id" : 999, "title" : "Le Petite Prince" }

> db.books.find({$text: {$search: 'prince'}}, {title:1})
{ "_id" : 145, "title" : "Azure in Action" }
{ "_id" : 999, "title" : "Le Petit Prince" }

```

用法语，只搜索到“Le Petit Prince”

用英语进行同样搜索，找到两本书

用英语搜索 prince，同时找到了法语和英语书

当搜索simultanment时，指定语言为法语，只能找到图书《Le Petit Prince》。当不指定语言搜索时——意味着使用默认的语言英语——会返回两本不同的图书。

怎么会这样？使用这个例子，你也许认为MongoDB会忽略与指定语言不一样的文档。但是如果看一下第三个find()，搜索“prince”单词，MongoDB就可以同时找到英语和法语的文档。

为什么会这样？答案取决于词根处理。当指定搜索字符串时，MongoDB会搜索关键字字符串的词根，而不是实际的字符串。相似的过程也会用到建立索引的过程中，这个过程也包括词根分析处理，而不是单词本身。因此，对于法语和英语，MongoDB使用词根处理的结果是不同的。

法语中，很容易看出MongoDB如何找到一个文档，因为图书描述包含单词“simultanment”。对于英语文档，原因相对没有这么明显。下面的列表可以帮助我们分析这个问题，而且演示了词根处理的一些限制。

列表9.6 “simulataneous”的词根分词结果

```
> db.books.find({$text: {$search: 'simultaneous'}},{title:1})
{ "_id" : 186, "title" : "Hadoop in Action" }
{ "_id" : 293, "title" : "Making Sense of Java" }
{ "_id" : 999, "title" : "Le Petite Prince" }
> db.books.find({$text: {$search: 'simultaneous',
    $language: 'french'}},{title:1})
>
```

用英语搜索 simultaneous

用法语搜索 simultaneous，没有结果

这个列表中，使用法语和英语同时搜索了单词“simultaneous”。正如我们期望的，用英文搜索时，会找到两本与“simultantment”相关的书。但是如果用法语搜索“simultaneous”，就无法找到相关的书。

不幸的是，此时，MongoDB法语词根处理“simultantment”的结果与英语词根处理的结果不同。结果有点迷惑，词根分析的过程不是精确的科学。但是在绝大多数情况下，我们会找到期望的结果。

幸运的是，语言对于分词的影响简单得多。对于分词，MongoDB可以使用某个语言特定的分词词典来进行分词。因此，语言对于分词的影响就比较简单。列表9.7是一个简单的例子。

列表9.7 语言如何影响分词。

```
> db.books.find({$text: {$search: 'de'}},{title:1})
{ "_id" : 36, "title" : "ASP.NET 4.0 in Practice" }
{ "_id" : 629, "title" : "Play for Java" }
{ "_id" : 199, "title" : "Doing IT Right" }
{ "_id" : 10, "title" : "OSGi in Depth" }
{ "_id" : 224, "title" : "Entity Framework 4 in Action" }
{ "_id" : 761, "title" : "jQuery in Action, Third Edition" }
> db.books.find({$text: {$search: 'de', $language: 'french'}}).count()
0
```

只在英语图书里搜索“de”

只在法语图书中搜索“de”

在这个例子里，先用英语查询单词“de”，然后使用法语搜索。当搜索英语完成后，会发现很多图书。此时，我们会找到作者名字里包含“de”的图书。找不到法语图书，因为法语里“de”是个分词，因此不会建立索引。

如果使用法语执行相同的搜索，不会找到结果，因为法语里“de”是个分词。解析后的字符串不会包含分词，因此无法搜索出结果。

正如我们看到的，语言对于文本搜索的结果有很大的影响，还会影响索引的创建过程。这也是为什么在使用索引、文档和关键字搜索时要指定语言的原因。如果只关心英语，任务就简单得多了。但是如果要考虑其他语言，则可以查阅MongoDB支持的语言。希望可以找到我们需要的语言。

9.6.4 可用的语言

MongoDB支持许多语言,而且这个列表还在持续增长。下面的列表是MongoDB 2.6(MongoDB 3.0也一样)支持的语言列表。下面列表还显示了可以使用的2个字母的语言简称:

- | | |
|-----------------|------------------|
| ■ da——danish | ■ no——norwegian |
| ■ nl——dutch | ■ pt——portuguese |
| ■ en——English | ■ ro——romanian |
| ■ fi——finnish | ■ ru——russian |
| ■ fr——french | ■ es——spanish |
| ■ de——german | ■ sv——swedish |
| ■ hu——hungarian | ■ tr——turkish |
| ■ it——italian | |

除了这个列表,我们还可以指定none。这样做时, MongoDB会跳过词根处理和分词。例如,某个文档使用的语言为none,它的索引会为每个单词创建入口。只有精确的单词才会建立索引,而不会进行分词处理,而且都不会排除分词单词。这个方法在文档包含用MongoDB处理有困难的单词时非常有用。坏处就是无法使用词根处理机制来搜索相似的单词,只能包含精确单词的搜索结果。

9.7 总结

Summary

正如我们看到的, MongoDB为数据库提供了丰富的基本文本搜索的功能。聚合框架提供了更复杂的搜索功能。但是MongoDB文本搜索有自己的局限性,而且并不能取代Elasticsearch或者Solr这样的专业搜索引擎。如果可以使用MongoDB的文本搜索功能,那么可以节约维护的复杂性,不需要在搜索引擎里也维护一份复制的数据。

现在,既然我们已经学习了MongoDB完整的搜索、更新和创建索引的功能,那么现在可以继续深入MongoDB 3.0的新特性了,而且新内容与MongoDB存储、更新和读取数据的底层机制相关: WiredTiger全新存储引擎!

WiredTiger与可拔插存储

WiredTiger and pluggable storage

本章内容

- WiredTiger
- 可拔插存储引擎
- MMAPv1 与WiredTiger 对比

在MongoDB v3.0之后, MongoDB主要的改变之一就是引入了可拔插存储引擎API。在本章里, 我们会看下什么是可拔插存储引擎, 以及为什么要加入MongoDB中。将会介绍与MongoDB捆绑的可拔插存储引擎WiredTiger, 并与MongoDB 3.0之前的默认存储引擎比较。还会比较两个引擎的速度、磁盘使用率和延迟等指标。还会介绍其他几个可拔插存储引擎, 它们可以成为替代选择。对于高级读者, 我们会揭秘可拔插存储引擎背后的底层原理。

10.1 可拔插存储引擎 API

Pluggable Storage Engine API

应用程序接口(API)是一组相对严格的构建软件应用使用的例程、协议和工具。例如, MongoDB提供了API允许其他的软件和MongoDB数据库进行交互, 而不需要MongoDB shell; 每个MongoDB驱动都使用了这些标准的API提供的功能。它们允许应用程序与MongoDB数据通信, 并对数据库中的文档数据执行CRUD操作。存储引擎是数据库和硬件直接的接口。存储引擎不会改变shell或驱动里执行的查询, 它也不会集群级别干扰MongoDB。但是存储引擎影响如何从磁盘写入、删除和读取数据, 还有存储数据使用的数据结构。

可拔插存储引擎API允许第三方为MongoDB开发存储引擎。在可拔插存储引擎API之前唯一可用的MongoDB存储引擎就是MMAPv1。

MongoDB仍然在使用MMAPv1存储引擎, 而且这也是3.0版本以后默认的存储引擎。MMAPv1存储引擎基于内存映射而且是MongoDB稳定的解决方案。当有大量要存储的数据时会发现MMAPv1的一个缺点, 既随着数据的增长它会快速消耗大量的磁盘空间, 每次预分配2GB的增长空间。许多数据库都支持预分配空间机制, MongoDB也不例外, 每个增量都是预分配2GB,

作为系统管理员要注意这个问题。

数据库管理员必须选择不同的存储引擎来决定如何在磁盘上存储数据。从3.0版本以后，可以告诉MongoDB使用哪个存储引擎，这也是可拔插存储引擎API的工作。它提供了MongoDB存储数据所需要的功能。MongoDB v3.0在MMAPv1之外提供了一个备选的存储引擎WiredTiger。我们会介绍更多关于WiredTiger存储引擎的知识，以及如何切换使用它来存储数据。但是首先我们要理解为什么MongoDB提供了不同的存储引擎。

我们来思考两种不同的应用：

- 新闻网站，比如Huffington邮报；
- 社交媒体，比如Twitter 或者 Facebook。

新闻网站可以阅读新闻。Huffington邮报平均每天发布1200条新闻，而且要被全球各地数千万的读者阅读^[1]。

与之对比是社交媒体网站，这里的用户之间分享各自的故事，这些故事比新闻短小得多。Twitter的推文140字左右，Facebook或Google+的状态也很短小。不同的使用情况对于存储引擎有不同的需求，如表10.1所示。

与社交网站相比，新闻网站不需要处理太多的数据，对读者来说，前端UI几乎是一样的。而社交网站，换句话说，必须处理百万级的推文或状态更新。每个访问者都有自己的组织，这要求显示的推文和更新必须与之相关。除了为不同的访问者发送不同的状态更新信息，社交媒体平台也需要存储每天几百万的新推文^[2]。

表 10.1 不同的情况/用户不同的需求

	新闻网站	社交媒体网站
文档更新	几百篇文章	几百万更新
平均大小	几千字节	T 字节
动态内容	内容不变	基于用户分发内容

新闻网站，同一时刻需要为不同的读者查询相同的新闻数据。许多数据库都提供了查询缓存，可以加速数据的查询过程。这种类型的新闻网站也可以使用外部缓存系统，比如Memcached或Redis来缓存高并发访问的数据。但是这些解决方案对于每个读者每次查询数据都不一样的社交网站帮助不大。此种应用需要不同的存储系统，从大规模数据集中过滤查询数据时提供更高的性能。社交网站也需要一个出色的写入性能存储系统来支持每天几百万的新闻存储需求。新闻网站没有这种性能问题，因为它们的写入操作只有几千个。

为了满足不同的系统需求，MongoDB实现了可拔插存储引擎，这样数据库管理员或者系统工

^[1]根据 2013 文章 DigiDay: <http://digiday.com/publishers/whos-winning-at-volume-in-publishing/>。
^[2]根据 Domo 的统计，2014 年，Facebook 每分钟有 240 万条内容更新，Twitter 有 27 万条推文。这个数量换算过来就是每天超过 35 亿次分享以及每天 4 亿条推文。

工程师可以选择符合自己需求的最佳性能的存储引擎。

下一节里我们会介绍一个与MongoDB捆绑发布的存储引擎：WiredTiger。

10.2 WiredTiger

WiredTiger是个高性能的、可伸缩的、开源的数据存储引擎，它专注于多核伸缩性和最佳的内存使用。多核伸缩可以通过使用现代编程技术实现，比如风险指针(hazard pointer)^[1]和无锁算法(lock-free)^[2]。

WiredTiger由Michael Cahill和Keith Bostic开发，他们作为架构师就职于Sleepycat Software公司，Bostic和其妻子创立该公司。在Sleepycat软件公司，他们设计并开发了Berkeley DB，世界上最广泛使用的嵌入式数据管理软件。

10.2.1 切换到 WiredTiger

在使用WiredTiger之前，要确保系统是64位系统，因为这是必须的。绝大部分新的计算机设备应该都支持64位系统。当设置MongoDB使用WiredTiger时，要记住要在新的dbPath目录下保存配置信息来启动MongoDB服务器。如果使用dbPath参数启动MMAPv1引擎的MongoDB服务，则是不会成功。这是因为MMAPv1存储结构与WiredTiger不兼容，还没有自动转换的机制。但是可以把基于MMAPv1的数据库迁移到WiredTiger数据库中，相反也可以使用mongodump和mongoexport实现。第13章会介绍更多关于dump创建数据镜像和恢复数据库的知识。

要在MongoDB里启用WiredTiger引擎，只需在默认的配置文件的YAML里设置一下存储参数即可（参考附录A里关于YAML的介绍），如下所示：

```
storage:
  dbPath: "/data/db"
  journal:
    enabled: true
  engine: "wiredTiger"
  wiredTiger:
    engineConfig:
      cacheSizeGB: 8
      journalCompressor: none
      collectionConfig:
        blockCompressor: none
      indexConfig:
```

^[1]在多线程编程中，很重要的问题就是追踪拿下被线程访问的内存块，并且判断是否有线程在访问它。风险指针就是这些被线程访问的内存块的指针列表，只要它们还在危险指针的列表中，就禁止其他线程修改或者删除指针和内存块。

^[2]多线程编程中，资源锁是非常重要的概念。无锁算法（Lock-free algorithm）是一种可以避免程序阻塞的编程模式，因为几个线程会彼此等待释放锁，并且确保程序作为一个整体继续工作。

prefixCompression: false

这是MongoDB一个基本的无压缩的启用WiredTiger 引擎的配置例子。表10.2展示了参数信息。

表 10.2 MongoDB 配置文件的不同参数

参数名称	描 述
dbPath	数据库文件存储的路径，默认/data/db
journal.enabled	是否启用日志。推荐启用，因为它可以保存断电期间还没有同步到磁盘的数据。64 位系统上默认是 true
engine	使用哪个引擎。默认是 mmapv1。要使用 WiredTiger，设置为 wiredTiger 即可
wiredTiger	这是 wiredTiger 专用的参数设置
engineConfig.cacheSize	这是 WiredTiger 引擎需要在内存中使用的内存大小，可以加速数据查询。默认是物理内存的一半，最少是 1GB
engineConfig.journalCompressor	告诉 WiredTiger 引擎使用哪种日志压缩器。默认是 snappy，但是也可以设置为 none 以获取最佳性能
collectionConfig.blockCompressor	告诉 WiredTiger 引擎使用何种压缩器压缩集合数据。有 3 种选择: none, snappy, zlib。我们会看到针对 3 个选项的测试对比，默认是 snappy
indexConfig.prefixCompression	告诉 WiredTiger 引擎是否为索引使用压缩。默认是 true

10.2.2 迁移数据到 WiredTiger

因为无法使用MMApV1格式的数据目录来启动使用WiredTiger引擎的MongoDB数据库，所以现在我们需要把数据库迁移到WiredTiger数据库中。可以通过创建镜像并恢复数据库来实现（第13章里有更多的信息）：

(1) 创建一个基于MMApV1引擎的MongoDB数据库镜像：

```
$ mkdir ~/mongo-migration
$ cd ~/mongo-migration
$ mongodump
```

这样就会在主目录下面创建一个mongo-migration文件夹。

(2) 停止mongod实例，确定没有mongod在运行状态：

```
$ ps ax | grep mongo
```

(3) WiredTiger更新MongoDB配置，正如之前描述的一样。

(4) 迁移基于MMAPv1引擎的数据库。假设dbPath路径是/data/db:

```
$ sudo mv /data/db /data/db-mmappv1
```

(5) 创建新的文件夹，然后给与MongoDB账号写入权限:

```
$ mkdir /data/db  
$ chown mongodb:mongodb /data/db  
$ chmod 755 /data/db
```

(6) 启动基于WiredTiger引擎的MongoDB实例。

(7) 把镜像导入基于WiredTiger引擎的MongoDB数据库:

```
$ cd ~/mongo-migration  
$ mongorestore dump
```

现在新的数据库在新的存储引擎下运行了。如果检查基于WiredTiger引擎的存储目录/data/db，和之前的基于MMAPv1引擎的目录/data/db-mmappv1对比，就会发现许多不同点。首先，如果数据库很大，就会注意到两个目录在磁盘使用方面的差别很大。我们会在下一节看到具体的测试对比。

要把基于WiredTiger引擎的数据库转换回基于MMAPv1引擎的，就重复以上过程，但是我们需要生成WiredTiger存储的景象，停止实例，修改配置为MMAPv1，使用此配置启动MongoDB，然后把数据镜像导入到基于MMAPv1引擎的数据库里。

另外一个方法是使用WiredTiger配置来运行第二个MongoDB实例，并且添加这个实例到可复制集，与现在的MMAPv1实例一起并存。第11章里会介绍可复制集群。

10.3 与 MMAPv1 对比

Comparison with MMAPv1

MongoDB的WiredTiger存储引擎的性能与MMAPv1引擎的相比会怎么样?

事实上，我们会使用Javascript和Shell脚本对比测试3组WiredTiger和MMAPv1引擎。我们会测试下面的配置:

- 默认的MMAPv1配置;
- 正常的WiredTiger，无压缩模式;
- 带snappy压缩模式的WiredTiger;
- 带zlib压缩模式的WiredTiger。

Zlib和snappy都是压缩算法。前者是DEFLATE压缩算法的抽象，它是LZ77算法的一个变种，而LZ77使用霍夫曼编码。Zlib在许多软件平台中很常见，而且是gzip压缩算法的基础。Snappy

由Google开发,并且广泛应用于Google项目中,比如BigTable。Snappy是个更中级的解决方案,它并非关注最大化压缩,而是关注告诉和合理的压缩比例。

我们会先测试数据库的插入性能,然后使用这个数据库来测试读取性能。在接下来的测试小节里,要记住很难从原始的测试结果中得出正确的结论。因为测试数据集的规模比实际项目的数据规模小得多,真实的项目里,我们会使用很多不同的搜索过滤器来查询数据,而实际项目中的数据结构复杂得多。本次测试只是简单的例子,让大家掌握如何使用现有的数据和过滤器来测试MongoDB的深入知识。

10.3.1 配置文件

为了能对比不同的存储引擎配置的磁盘使用情况,就要使用不同的数据库路径存储配置文件。在生产环境下,会在机器上使用公共的MongoDB安装路径。

MMAPv1实例的配置文件叫做mmapv1.conf,如下所示,使用了YAML格式:

```
storage:
  dbPath: "./data-mmapv1"
  directoryPerDB: true
  journal:
    enabled: true
    systemLog:
      destination: file
      path: "./mongodb-server.log"
    logAppend: true
    timeStampFormat: iso8601-utc
net:
  bindIp: 127.0.0.1
  port: 27017
  unixDomainSocket:
    enabled: true
```

对于WiredTiger配置,除了存储部分,与之前的代码类似。对于无压缩版本,可以为WiredTiger引擎使用下面的配置,命名为wiredtiger-uncompressed.conf:

```
storage:
  dbPath: "./data-wt-uncompressed"
  directoryPerDB: true
  journal:
    enabled: true
  engine: "wiredTiger"
  wiredTiger:
    engineConfig:
      cacheSizeGB: 8
      journalCompressor: none
    collectionConfig:
      blockCompressor: none
    indexConfig:
      prefixCompression: false
```

对于snappy压缩情况，可以使用下面的配置，它的不同点在于黑字体部分。文件命名为“wiredtiger-snappy.conf”：

```
storage:
  dbPath: "./data-wt-zlib"
  directoryPerDB: true
  journal:
    enabled: true
    engine: "wiredTiger"
  wiredTiger:
    engineConfig:
      cacheSizeGB: 8
      journalCompressor: none
    collectionConfig:
      blockCompressor: snappy
    indexConfig:
      prefixCompression: true
```

最后，对于zlib实例，会使用下面的存储配置，命名为wiredtiger-zlib.conf:

```
storage:
  dbPath: "./data-wt-snappy"
  directoryPerDB: true
  journal:
    enabled: true
    engine: "wiredTiger"
  wiredTiger:
    engineConfig:
      cacheSizeGB: 8
      journalCompressor: none
    collectionConfig:
      blockCompressor: zlib
    indexConfig:
      prefixCompression: true
```

如果我们按照传统方式安装MongoDB，这些配置就会在configs文件夹下。我们可以一起使用这些配置，例如，为MMAPv1运行下面的配置：

```
$ bin/mongod --config configs/mmapv1.conf &
```

这个命令可以在后台使用默认的配置来运行MongoDB实例。

10.3.2 插入脚本与基准测试脚本

我们会使用下面的JavaScript代码来插入数据到测试数据库，文档有4个不同数据类型的字段、1个数组字段，包含4种不同数据类型的8个文档元素。这不是真实的情况，而且压缩算法在更加复杂的数据集中的工作过程差距很大：

```
for (var j = 0; j < 10000; j++) {
  var r1 = Math.random();

  // A nice date around year 2000 年的某一天
```

```

var dateFld = new Date(1.5e12 * r1);
var intFld = Math.floor(1e8 * r1);
// A nicely randomized string of around 40 characters 40 个字符的随机字符串
var stringFld = Math.floor(1e64 * r1).toString(36);
var boolFld = intFld % 2;

doc = {
  random_date: dateFld,
  random_int: intFld,
  random_string: stringFld,
  random_bool: boolFld
}

doc.arr = [];

for (var i = 0; i < 16; i++) {
  var r2 = Math.random();

  // A nice date around year 2000 2000 年的某一天
  var dateFld = new Date(1.5e12 * r2);
  var intFld = Math.floor(1e8 * r2);
  var stringFld = Math.floor(1e64 * r2).toString(36);
  var boolField = intFld % 2;

  if (i < 8) {
    doc.arr.push({
      date_field: dateFld,
      int_field: intFld,
      string_field: stringFld,
      bool_field: boolFld
    });
  } else {
    doc["sub" + i] = {
      date_field: dateFld,
      int_field: intFld,
      string_field: stringFld,
      bool_field: boolFld
    };
  }
}

db.benchmark.insert(doc);
}

```

这个JavaScript代码段存储在insert.js中，将要在测试数据库里插入10 000个文档。这个脚本会使用下面的批处理脚本文件来运行4个配置，为不同的配置做16次相同的插入工作：

```
#!/bin/bash
```

```
export MONGO_DIR=/storage/mongodb
export NUM_LOOPS=16
```

```
configs=(
  mmapv1.conf
  wiredtiger-uncompressed.conf
  wiredtiger-snappy.conf
  wiredtiger-zlib.conf
)
```



```

)

cd $MONGO_DIR
for config in "${configs[@]"; do
    echo "==== RUNNING $config ====="
    echo "Cleaning up data directory"
    DATA_DIR=$(grep dbPath configs/$config | awk -F\" '{ print $2 }')
    rm -rf $MONGO_DIR/$DATA_DIR/*

    echo -ne "Starting up mongod... "
    T="$(date +%s)"
    ./bin/mongod --config configs/$config &

    # wait for mongo to start
    while [ 1 ]; do
        ./bin/mongostat -n 1 > /dev/null 2>&1
        if [ "$?" -eq 0 ]; then
            break
        fi
        sleep 2
    done
    T="$((date +%s)-T)"
    echo "took $T seconds"

    T="$(date +%s)"
    for l in $(seq 1 $NUM_LOOPS); do
        echo -ne "\rRunning import loop $l"
        ./bin/mongo benchmark --quiet insert.js >/dev/null 2>&1
    done
    T="$((date +%s)-T)"

    echo
    echo "Insert performance for $config: $T seconds"

    echo -ne "Shutting down server... "
    T="$(date +%s)"
    ./bin/mongo admin --quiet --eval "db.shutdownServer({force: true})" >/dev/null 2>&1

    while [ 1 ]; do
        pgrep -U $USER mongod > /dev/null 2>&1
        if [ "$?" -eq 1 ]; then
            break
        fi
        sleep 1
    done
    T="$((date +%s)-T)"
    echo "took $T seconds"

    SIZE=$(du -s --block-size=1 $MONGO_DIR/$DATA_DIR | cut -f1)
    SIZE_MB=$(echo "scale=2; $SIZE/(1024*1024)" | bc)
    echo "Disk usage for $config: ${SIZE_MB}MB"
done

```

这个脚本假设MongoDB安装在bin文件夹下，与脚本的目录文件夹一样，使用给定的配置参数来启动MongoDB实例。因此，在运行脚本之前应该确保停止了MongoDB实例。

10.3.3 插入测试结果

在运行这个批处理脚本之后，会看到如下的输出。每个机器的实际时间可能不同，取决于操作系统和内核。此例子使用的硬件是四核i5-3570K，3.4 GHz主频，运行内存16 GB，而且每个存储使用了ext4格式化的LVM分区，分布在2个Hitachi Deskstar T7K250 250 GB磁盘上，SATA模式，每分钟7200转。系统是Ubuntu 14.04，内核是Linux kernel 3.13.0，结合Intel 64-bit 架构：

```
===== RUNNING mmapv1.conf =====
Cleaning up data directory
Starting up mongod... took 102 seconds
Running import loop 16
Insert performance for mmapv1.conf: 105 seconds
Shutting down server... took 1 seconds
Disk usage for mmapv1.conf: 4128.04MB
===== RUNNING wiredtiger-uncompressed.conf =====
Cleaning up data directory
Starting up mongod... took 2 seconds
Running import loop 16
Insert performance for wiredtiger-uncompressed.conf: 92 seconds
Shutting down server... took 3 seconds
Disk usage for wiredtiger-uncompressed.conf: 560.56MB
===== RUNNING wiredtiger-snappy.conf =====
Cleaning up data directory
Starting up mongod... took 1 seconds
Running import loop 16
Insert performance for wiredtiger-snappy.conf: 93 seconds
Shutting down server... took 2 seconds
Disk usage for wiredtiger-snappy.conf: 380.27MB
===== RUNNING wiredtiger-zlib.conf =====
Cleaning up data directory
Starting up mongod... took 2 seconds
Running import loop 16
Insert performance for wiredtiger-zlib.conf: 104 seconds
Shutting down server... took 3 seconds
Disk usage for wiredtiger-zlib.conf: 326.67MB
```

服务启动、插入数据的时间，以及关闭进程、衡量关闭后磁盘的使用情况可以参见表10.3。

表 10.3 对比 MMAPv1 和 WiredTiger 操作

	MMAPv1	WT	WT snappy	WT zlib
启动	102 sec	2 sec	1 sec	2 sec
插入工作	105 sec	92 sec	93 sec	104 sec
关闭	1 sec	3 sec	2 sec	3 sec
磁盘使用	4128.04 MB	560.56 MB	380.27 MB	326.67 MB

WiredTiger启动服务器和初始化存储目录的过程非常痛苦。但是这是一次性过程，因此不是一个关键的衡量标准。另外一个针对MongoDB启动与关闭的测试使用了预先初始化的存储目录，MMAPv1启动耗时4秒。我们会在下一节里测试读取的性能。

当使用WiredTiger存储引擎时对于插入还有一些好处，处理比例子更大的数据集时差别会更明显。还要记住，这个测试不会同时使用多个Mongo客户端连接运行插入工作。

这些测试结果最显著的一点就是磁盘的使用：无压缩模式的WiredTiger比MMAPv1引擎的MongoDB少用15%的磁盘，如果加入压缩，会少10%！如果使用snappy压缩配置会在插入速度和磁盘使用之间找到一个好的平衡点。这几乎与无压缩模式的WiredTiger一样的速度，但是仍然在这个数据集上有180MB。Zlib有更好的压缩比，但是此模式插入会耗费10秒。

10.3.4 读性能测试脚本

目前我们已经测试了不同存储配置的写入性能，但是我们对于某些程序的读性能更感兴趣。

这里是一段简单的Javascript代码，从测试集里查询所有的纪录，并且顺序遍历每个文档。注意，它没有测试搜索和过滤，因为需要提前为搜索定义值的集合：

```
c = db.benchmark.find();
while(c.hasNext()) c.next();
```

这些JavaScript代码放在read.js文件里，而且通过下面的read.sh shell脚本运行，与插入数据的脚本类似：

```
#!/bin/bash

export MONGO_DIR=/storage/mongodb
export NUM_LOOPS=16

configs=(
    mmapv1.conf
    wiredtiger-uncompressed.conf
    wiredtiger-snappy.conf
    wiredtiger-zlib.conf
)

sudo echo "Acquired root permissions"
cd $MONGO_DIR
for config in "${configs[@]"; do
    echo "===== RUNNING $config ====="
    echo "Clearing memory caches"
    sync
    echo 3 | sudo tee /proc/sys/vm/drop_caches

    echo -ne "Starting up mongod... "
    T="$(date +%s)"
    ./bin/mongod --config configs/$config &

    # wait for mongo to start
    while [ 1 ]; do
        ./bin/mongostat -n 1 > /dev/null 2>&1
        if [ "$?" -eq 0 ]; then
            break
        fi
    done
```

```

    sleep 2
done
T="$((date +%s)-T))"
echo "took $T seconds"

rm -f timings-${config}.txt
T="$date +%s"
for l in $(seq 1 $NUM_LOOPS); do
    echo -ne "\rRunning read loop $l"
    /usr/bin/time -f "%e" -o timings-${config}.txt -a --quiet ./bin/mongo
benchmark --quiet read.js >/dev/null 2>&1
done
T="$((date +%s)-T))"

echo
echo "Read performance for $config: $T seconds"
echo -ne "Shutting down server... "
T="$date +%s"
./bin/mongo admin --quiet --eval "db.shutdownServer({force: true})" >/
dev/null 2>&1

while [ 1 ]; do
    pgrep -U $USER mongod > /dev/null 2>&1
    if [ "$?" -eq 1 ]; then
        break
    fi
    sleep 1
done
T="$((date +%s)-T))"
echo "took $T seconds"
done

```

这个脚本也会计时启动和关闭MongoDB服务器进程的时间，所以可以看到它使用已经初始化的存储目录启动会快得多。因为脚本需要清理内存缓存以获得精确的查询时间——不是从缓存里获取数据，而是从存储系统本身获取——需要输入sudo的密码，这样它可以清空内存缓存。

10.3.5 读性能结果

当运行脚本时，会看到类似下面的输出结果：

```

===== RUNNING mmapv1.conf =====
Clearing memory caches
3
Starting up mongod... took 3 seconds
Running read loop 16
Read performance for mmapv1.conf: 33 seconds
Shutting down server... took 1 seconds
===== RUNNING wiredtiger-uncompressed.conf =====
Clearing memory caches
3
Starting up mongod... took 2 seconds
Running read loop 16
Read performance for wiredtiger-uncompressed.conf: 23 seconds

```

```

Shutting down server... took 2 seconds
===== RUNNING wiredtiger-snappy.conf =====
Clearing memory caches
3
Starting up mongod... took 3 seconds
Running read loop 16
Read performance for wiredtiger-snappy.conf: 21 seconds
Shutting down server... took 1 seconds
===== RUNNING wiredtiger-zlib.conf =====
Clearing memory caches
3
Starting up mongod... took 2 seconds
Running read loop 16
Read performance for wiredtiger-zlib.conf: 21 seconds
Shutting down server... took 1 seconds

```

每行中的“3”是从内存缓存清理缓存的执行脚本标志，所以可以忽略。现在可以清晰地看到，每种不同的MongoDB配置，其启动和关闭的时间差不多，如表10.4所示。

表 10.4 不同配置的启动和关闭时间

	MMAPv1	WT	WT snappy	WT zlib
启动	3 sec	2 sec	3 sec	2 sec
读取	33 sec	23 sec	21 sec	21 sec
关闭	1 sec	2 sec	1 sec	1 sec

MMAPv1配置读取数据耗费的时间比WiredTiger配置多10秒。如果检查计时文件，就会看到其原因。时间数据如图10.1所示。

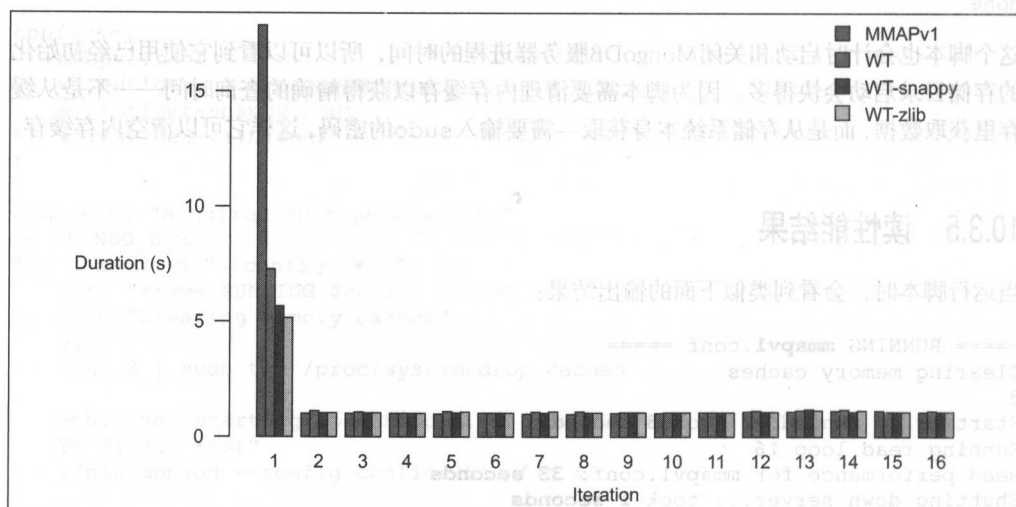


图 10.1 MMAPv1 和 WiredTiger 的读取性能

很明显，第一次迭代花费了最长的时间，因为每个后续的迭代都会从缓存里直接获取数据。对于纯读取，MMAPv1显然是最慢的。在纯读取时，压缩版本的WiredTiger有最好的性能。

但是对于缓存结果，MMAPv1比WiredTiger稍微快一点。图形表示的结果不够明显，但是计时文件很明确，内容如表10.5所示。

表 10.5 不同存储引擎的查询操作计时 (s)

	MMAPv1	WT	WT-snappy	WT-zlib
1	17.88	7.37	5.67	5.17
2	1.01	1.1	1.05	1.05
3	1	1.05	1.02	1.08
4	1.03	1.03	1.05	1.08
5	0.99	1.07	1.04	1.08
6	1.03	1.03	1.05	1.08
7	0.96	1.06	0.99	1.07
8	0.99	1.08	1.01	1.06
9	0.97	1.03	1.02	1.03
10	0.96	1.03	1.03	1.03
11	0.99	1.03	1.06	1.06
12	1.01	1.07	1.06	1.07
13	0.98	1.04	1.08	1.06
14	1.01	1.08	1.03	1.07
15	1.08	1.05	1.05	1.05
16	1.02	1.06	1.04	1.06

10.3.6 测试结论

WiredTiger比MMAPv1优秀多少？我们已经看了服务的启动和关闭时间、插入数千个平均大小文档和查询并取得这些文档的时间，也看到了存储目录的磁盘使用情况。

我们还没有测试多个MongoDB客户端连接同时发送请求的情况，也没有测试随机搜索和过滤的性能。这是真实世界常见的例子，需要比现在更复杂的测试配置。我们希望这个测试例子可以给大家带来测试MongoDB其他方面性能的提示。

从本章的测试结果可以得出结论：可以从磁盘使用中获取更多的收益。对于小规模应用，非常关注资源的使用，这是决定性因素，而且我们应该使用压缩版本的WiredTiger。Zlib版本可以获得最好的性能和开销比率。对于不关注存储开销的应用系统，无压缩模式的WiredTiger存储引擎配置是最佳的选择。如果需要压缩，可以使用snappy压缩算法，它会比zlib配置获得稍微好点的速度。

即使磁盘存储空间对于企业用户来说都不是问题，但是读取速度确实个非常重要的因素。尤其是社交网站，每个访问者都有专门的过滤器，所以经常发生丢失缓存的问题。

再强调一次，本章里的测试不能全部代表真实的情况，因此不能强行从本次测试的数据中得出绝对的结论。但是我们希望这些测试可以给大家一个基本的指导意见，可以根据实际的应用需求来调整测试脚本以满足需求。这样，我们可以得出更好的结论来决定哪个存储引擎更适用于特定的用例场景。

还有几个其他依赖于硬件和OS系统配置的环境因素要考虑。虽然没有在本章里讨论，但是也可能影响测试结果的性能。当比较存储引擎时，应该固定环境参数。虽然某些存储引擎在特定的操作系统下可能获取更好的性能，但是其他系统可能会降低性能。因此，我们在得出结论前先限制测试的前提条件。

10.4 其他可拔插存储引擎的例子

Other examples of pluggable storage engines

前面一节我们讨论了WiredTiger引擎，以及与MMAPv1做了磁盘使用和读/写速度的测试对比。本节里我们会介绍几个其他可用的存储引擎。

其中一个例子是RocksDB，它由Facebook开发，基于Google的LevelDB数据库，受到Apache的Hbase启发。RocksDB是个键值对存储库，最大化开发闪存和内存提供高速的读/写潜力，通过LSM树引擎^[1]来提供低延迟数据库。这使得RocksDB成为高并发写入性能应用很好的存储选择，比如社交媒体应用。MongoDB可以使用RocksDB，这样也可以使用它的高写入性能优势。更多关于如何实现的内容可以参考RocksDB的声明博客^[2]。

Tokutek的TokuFT（正式名字TokuKV）是另外一个键值存储库，可以使用分形树索引^[3]来替换WiredTiger默认的B-树数据结构。

在MongoDB v3.0之前，Tokutek维护了MongoDB的一个分支，叫TokuMX。它使用了自己的技术TokuFT作为存储引擎——只根据MongoDB的可拔插API，叫做TokuMXse^[4]，表示标准的存储引擎。虽然TokuMX分支集成了许多功能，比如集群索引以及通过更新MongoDB代码库来进行快速免费读更新操作，但是TokuMXse版本没有办法拥有它们，因为插件式存储引擎API是当前设计的。在这个意义上，TokuMXse将允许我们进行标准的MongoDB安装，带有可靠的高性能存储和压缩功能。

Tokutek已经被Percona收购，所以与TokuFT相关的开发工作都由Percona负责。Percona已经

^[1] 日志结构合并树 LSM（Log Structure Merge trees）在更新数据场景下性能优于 B-树。WiredTiger 团队对比了 B-树和 LSM 树：<https://github.com/wiredtiger/wiredtiger/wiki/Btree-vs-LSM>。更多关于 LSM 树的内容可以参考 https://en.wikipedia.org/wiki/Log-structured_merge-tree。

^[2] 阅读 <http://rocksdb.org/blog/1967/integrating-rocksdb-with-mongodb-2/> 获取更多信息。

^[3] 分形树索引是个 Tokutek 创新，在数据集增大的时候仍然可以保证数据写入性能的一致性。参考 <http://www.tokutek.com/resources/technology/>。

^[4] Tokutek 2015 年 1 月宣布：<http://www.tokutek.com/2015/01/announcing-tokumxse-v1-0-0-rc-0/>。

宣布了使用TokuMXse进行MongoDB的实验性开发^[1]。

最后一个例子就是InnoDB，MySQL 的用户比较熟悉。这个引擎被MySQL用来作为备选的MyISAM存储引擎。InnoDB使用了事务日志重放功能，当出现错误时，可以从失败日志中获取最新的操作并同步数据到磁盘上，这与WiredTiger和MMAPv1日志工作的原理类似。在出错之后，MyISAM必须要检查整个数据库来修复索引或者还没有持久化数据到磁盘的表。随着数据库规模的增长，这对于数据库的可用性有很大的影响。InnoDB没有这种问题，因此在更大数据库级别提供了更好的可用性。InnoDB技术并非MySQL的一部分。有个InnoDB存储引擎的分支叫XtraDB，被MariaDB使用，而且在以后的MongoDB中可能有针对InnoDB的模块。^[2]

10.5 高级主题

Advanced topics

本节里我们将会讨论更高级的主题，以便更好地掌握存储引擎如何工作，以及为什么有许多不同的存储引擎实现技术。本节介绍的知识点并非是基础的MongoDB如何使用存储引擎运行实例服务，而是更多背后的原理介绍。

10.5.1 可拔插引擎如何工作？

MongoDB源代码包含特殊的类来处理存储模块。在编写本书的时候，MongoDB源代码暴露了下面的类：

- StorageEngine——组成存储引擎基本结构的虚类；
- MMAPV1Engine——MMAPv1存储引擎插件类；
- KVStorageEngine——键值存储引擎类；
- KVEngine——类使用的键值引擎；
- MMAPV1DatabaseCatalogEntry——基于MMAPv1数据库的目录项目类；
- RecordStore——记录存储对象的基类；
- RecordStoreV1Base——MMAPv1记录存储对象的基类，MMAPV1DatabaseCatalogEntry使用了它；
- WiredTigerKVEngine——WiredTiger引擎；
- WiredTigerFactory——使用WiredTigerKVEngine类为键值引擎来创建

^[1] Percona 2015 年 5 月宣布：<https://www.percona.com/blog/2015/05/08/mongodb-perconatokumxse-experimental-build-rc5-available/>。

^[2] 【译者注】2008 年 Sun 以 10 亿美元价格收购了 MySQLAB 公司，2009 年 SUN 被 Oracle 收购，MySQL 落入 Oracle 公司，发展受制于商业策略，沦为鸡肋。2009 年，MySQL 的创始人 Michael Widenius 重新开始 MySQL 的一个分支开发 MariaDB，名称取自女儿名字 Maria。

KVStorageEngine对象的工厂类；

- WiredTigerRecordStore——WiredTigerKVEngine类使用的记录对象类型。

我们可以在图10.2里看到这些类型。^[1]

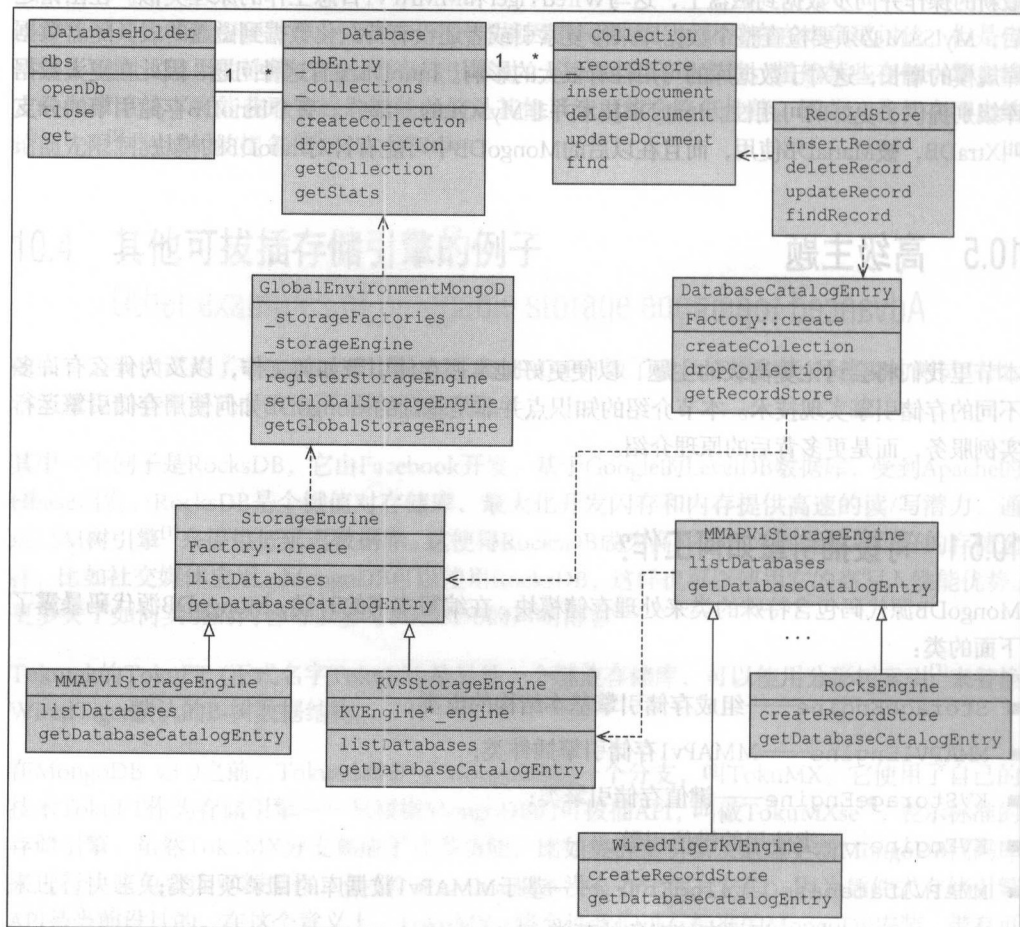


图 10.2 MongoDB 多存储引擎支持的机制

这些类型分为三层：

- 存储引擎，在顶层——作为MongoDB的存储插件接口。
- 存储管理，在中间层——在MMAPv1情况下它是目录项目类，在基于键值存储引擎的情况下是键值存储(KVEngine)。

^[1]【译者注】图 10.2 是 MongoDB 存储引擎源码的类图，表示各个类型之间的关系。

- 记录存储，在底层——它管理了实际的MongoDB数据，比如集合和数据项。在此级别，它通过与实际的存储引擎通信来执行CRUD操作。

这些层次组成了转换器，如果愿意，可以查看MongoDB的数据结构和存储引擎的工作原理。MongoDB专注于自己的数据结构——BSON——以及自己基于JavaScript的脚本语言，它被MongoDB驱动用来和MongoDB进行通信。

如何存储和查询数据的工作就交给了存储引擎。

10.5.2 数据结构

在MongoDB可拔插存储引擎API里，有针对键值存储系统的基类。键值存储系统是可以确保根据键值快速查询集合中数据的存储系统。我们可以告诉存储系统为快速查询而在某个字段上建立索引。

存储这种键值对数据的常用数据结构就是B-树，它也用在WiredTiger存储引擎里。B-树由Rudolf Bayer和Ed McCreight发明。他们就职于波音研究实验室（Boeing Research Labs）。也有传言说B-树中的B代表Bayer，使用Rudolf Bayer的名字命名，但是其他人说B代表波音公司（Boeing），是在波音实验室发明的。无论如何，最初的命名故事都没有办法考证了，我们更感兴趣的是它如何工作。

B-树的强大之处是基于磁盘的存储按块进行。存储在磁盘上的数据分块存储在文件中。在真实的项目里，这些块通常是4KB大小，这也是ExtFS和NTFS文件系统默认的大小。更大的块允许在磁盘上存储更大的文件。后面我们会讨论大于4TB的文件。

B-树存储系统使用了这些块作为存储数据元素索引的节点。每个节点可以保存4 096B的索引信息，而且这些索引信息是基本排序的。下面的例子中，假设每个节点可以保存3个索引。记住，在真实的项目中，数量可能更多。

B-树从根节点开始，在这个节点中，我们可以使用键值来存储数据记录。这些数据记录可能由指针来指向实际的值，或者也可能在数据记录里直接保存数据。更有趣的是如何查找这些记录，因为这是真正需要快速完成的工作。每个节点都有几个数据记录，通过索引的键值来建立索引，而且在每个记录中都有一个指针占位符区域可以指向其他的节点。值为12指针左侧的节点包含的值都会比12小。相似地，12和65指针之间的记录值都会介于12和65之间。这样，搜索的时候，算法就可以快速知道遍历的节点。

要想此算法高效地工作，B-树结构中的节点的每个索引的键值就必须是排序的。要优化磁盘搜索时间，推荐尽可能多地在节点上存储数据。节点大小与磁盘块的大小一样，真实项目中的大小是4 096B。

想像一下一个空的数据库，包含一个空的节点。在无序的集合中添加记录，一旦节点满了，

就需要创建新的节点。我们假设下面是要添加key的集合，没有排序：

{ 2, 63, 42, 48, 1, 62, 4, 24, 8, 23, 58, 99, 38, 41, 81, 30, 17, 47, 66 }

记住，我们使用了最大节点大小存放3个元素。对于图10.3所示键值结构，它是例子集合中的前三个元素，排序如下。

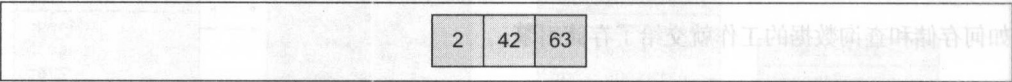


图 10.3 带有 3 个元素的根节点

当添加第4个元素48时，会在42和63之间创建新的节点，把48存储到新节点，因为48介于42和63之间。下一个元素1，在节点2之前，因为1小于2。然后下一个数62会保存到48节点里，因为62介于42和63之间。下面的3个键——4, 24, 8——会保存到2和42之间的节点里，因为这些数都介于2和42之间。在新的节点里，键值是排序存储的。在这些步骤之后，树的节点如图10.4所示。注意每个节点中数字的保存顺序。

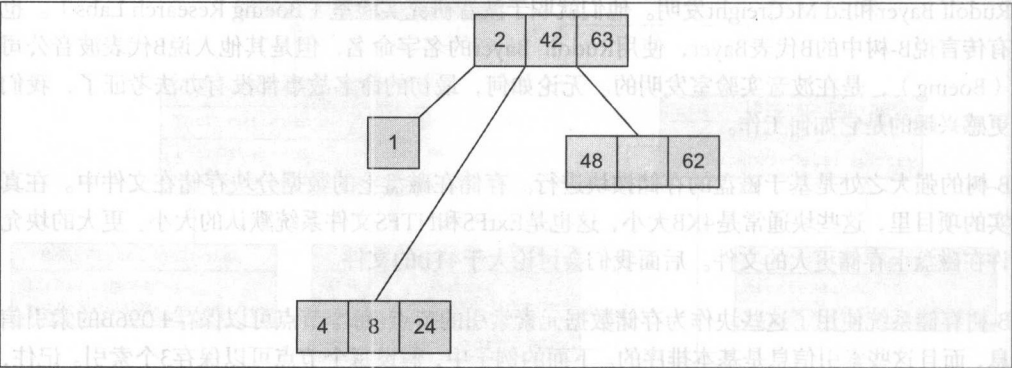


图 10.4 早期版本的 B-树

继续处理这个集合。接下的键值23，介于2和42之间，将会插入到包含4, 8, 24的节点之间，因为23介于8和24之间。然后接下来是58，它会加入到48和62之间。99会插入到63之后的一个新的节点中，因为99大于63。数值38，在2和42之间，将会访问包含4, 8, 24的节点，但是由于该节点已经满了，故它将会保存到24后的新节点里，因为38大于24。41会加入38所在的节点里。47介于42和63之间，所以它要被保存到包含48, 58, 62的节点中。因为节点已经满了，所以在48之前会继续深入创建新节点。最后，66将会插入到81和99所在的节点里，因为66大于63。最后的结构如图10.5所示，每个节点的值都是排序过的。

因为包含数据的节点是排序过的，所以就可以快速查找数据。它插入新数据的速度也很快，因为这和查询数据遍历的速度是一样的，直到找到空缺的数据槽，或者创建新的节点保存数据。

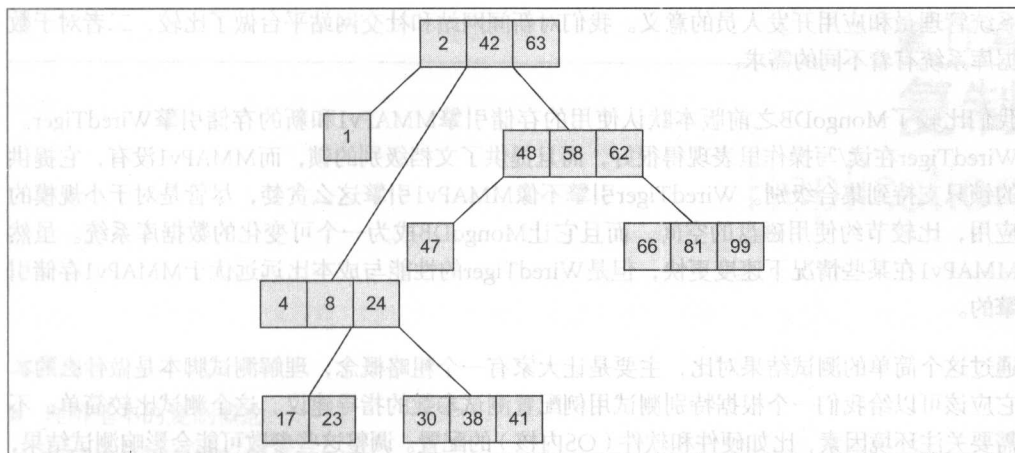


图 10.5 最终版本的 B-树

10.5.3 锁

在MongoDB早期版本中，每个连接都会有锁，在服务器级别，使用互斥锁（互相排斥），这种可以允许多个客户端或者多线程访问相同资源的机制。这种情况下的资源是数据库服务器。但是不是同时并行的。这是最坏的锁，尤其是需要大数据数据库引擎时，同时可能有几千个客户端并发访问。

在版本2.2以后做了改变，实现了数据库级别的锁。互斥锁应用在数据库级别而不是整个MongoDB服务器实例上，这是个重大的改进。但是MongoDB线程会尝试获取队列中的写入锁，并且使用串行方式处理它们，无论当前锁线程是否释放锁都可以允许线程继续工作。在小规模数据库里，这可能非常快，没有什么太大的性能影响；但在更大的数据库里，每秒几千次写入请求，这会成为致命问题，严重降低应用程序的性能。

这个问题在MongoDB v3.0之后被修复，此版本引入了集合级别的锁机制，对于MMAPv1存储引擎可用。这个新特性把锁机制进一步降低到集合级别。这意味着多个请求可以被同时处理，而不需要互相阻塞，只要它们可以写入到不同的结合中即可。

在MongoDB v3.0以后使用WiredTiger存储引擎，还可以支持文档级别的锁。这是一个更细粒度的锁——多个请求现在可以同时访问一个集合而不需要互相阻塞，只要没有同时写入一个文档。

10.6 总结

Summary

MongoDB v3.0引入了可拔插引擎的全新架构。在本章里，我们介绍了详细的内容，以及对于

系统管理员和应用开发人员的意义。我们对新闻网站和社交网站平台做了比较，二者对于数据库系统有着不同的需求。

我们比较了MongoDB之前版本默认使用的存储引擎MMAPv1和新的存储引擎WiredTiger。WiredTiger在读/写操作里表现得很好，而且提供了文档级别的锁，而MMAPv1没有，它提供的锁只支持到集合级别。WiredTiger引擎不像MMAPv1引擎这么贪婪，尽管是对于小规模的应用，比较节约使用磁盘的空间，而且它让MongoDB成为一个可变化的数据库系统。虽然MMAPv1在某些情况下速度更快，但是WiredTiger的性能与成本比远远优于MMAPv1存储引擎的。

通过这个简单的测试结果对比，主要是让大家有一个粗略概念，理解测试脚本是做什么的，它应该可以给我们一个根据特别测试用例配置测试参数的指导建议。这个测试比较简单，不需要关注环境因素，比如硬件和软件（OS内核）的配置。调整这些参数可能会影响测试结果，所以当比较不同的存储引擎时要格外小心。

我们还介绍了一些其他的存储引擎平台，比如RocksDB和TokuFT，两个都在实验阶段，但是提供了非常有趣的特性，因为RocksDB基于LSM树，而TokuFT基于分形树。看看它们如何在不同的情况下使用WiredTiger引擎进行工作，将会非常有趣。

我们也看了一些设计存储引擎时考虑的高级概念。最大化读/写性能，最小化磁盘I/O，而且在更高级别情况下使用智能并发锁机制来提供最佳的并发，这种方案在开发存储引擎的时候扮演着至关重要的作用。

理论已经够了！下一章的主题是实际项目中会用到的内容——复制。

本章内容

- 理解基本的复制概念
- 驱动连接到可复制集合
- 管理可复制集合和处理灾备
- 使用写入关注点增加写入持久性
- 使用阅读偏好优化读取性能
- 使用标签管理复杂的可复制集合

复制是绝大多数数据库管理系统的中心任务，因为一个不可避免的事实是发生故障。如果要想生产数据在发生故障后依然可用，就需要确保为生产数据库必须多部署一台机器。复制提供了数据保护、高可用和灾难恢复的机制。

本章我们将先介绍复制的基础概念以及它的使用场景，然后通过深入介绍可复制集来介绍MongoDB的复制功能。最后，我们将会介绍如何使用驱动去连接MongoDB的复制集群，如何使用写入关注点以及如何使用负载均衡来跨复制集群节点以读取数据。

11.1 复制概览

Replication overview

复制是跨多个MongoDB服务器(节点)分布和维护数据的方法。MongoDB可以把数据从一个节点复制到其他节点并在修改时进行同步。这种类型的复制通过一个叫可复制集的机制提供。集群中的节点配置为自动同步数据，并且在服务器出错时自动灾备。MongoDB也提供对于旧的复制方法的支持。这个旧方法叫做主从模式，现在已经过时了，但是主从复制仍然可以在MongoDB 3.0里使用。两种方法类似，主节点接受所有的写请求，然后所有的从节点读取，并且异步同步所有的数据。

主从复制和可复制集群使用了相同的复制机制，但是可复制集群额外增加了自动化灾备机制：

如果主节点宕机，无论什么原因，其中一个从节点会自动提升为主节点。除此之外，可复制集群还提供了其他改进，比如更易于恢复和更复杂地部署拓补网络。因此我们很少再用简单的主从复制机制^[1]。可复制集群是推荐的生产环境下的复制策略，我们将会在本章里重点解释底层原理和实战可复制集群例子，只简要介绍主从复制机制。

当然，理解复制的缺点也非常重要，最重要的是回滚机制。在可复制集群里，数据并非真正地提交，直到它被写入大多数集群的节点中，这指的是50%上的服务器。因此，如果可复制集群只有两个服务器，就意味着没有服务器可以停机。如果主节点在复制数据之前停机，其他成员将会继续接受写入，而且任意未复制的数据必须回滚，意味着它不能被读取。我们会在后面详细介绍这种场景的问题。

11.1.1 为什么复制很重要

所有的数据库都无法摆脱环境故障的影响。复制提供了一种防备故障的方式。我们讨论的是什么类型的故障呢？这里我们列举了一些常见情况：

- 应用程序和数据库之间的连接丢失了。
- 计划内停机阻止了服务器按照预期上线。

绝大部分主机都提供了意外停机重启机制，这种停机是无法预测的。简单的重启都会耗费几分钟时间，让数据库重新上线。但是当重启时我们应该怎么办？例如，新安装软件或者硬件阻止了MongoDB或者操作系统正常启动。

- 电源停电。虽然绝大多数现代化机房都提供了冗余电源设备，但是也没有办法阻止数据中心的人为错误或者局部限制用电导致的数据库停机。
- 数据库硬盘驱动器故障。硬盘驱动寿命一般有几年，而且故障失败率超出我们的想像^[2]。虽然我们对于MongoDB数据库偶尔的故障可以接受，但是应该无法接受丢失数据。所以最好有个数据副本，这就是复制机制提供的。

除了避免故障带来的损失意外，复制对于MongoDB的持久化功能也非常重要。当无日志运行MongoDB时，MongoDB数据文件无法保证在遇到意外关机时保证数据文件不冲突——启用日志可以保证数据文件不冲突。没有日志，如果发生单节点关机故障，复制机制应该可以保证复制干净的数据文件。

当然，复制机制最好也使用日志功能。毕竟，我们仍然希望高可用和快速灾备。在这种情况下，日志可以加快恢复工作，因为它允许通过简单的重放日志就可以让失败的节点重新启动上线。这比从一个存在的复制节点上同步数据再恢复服务快得多。

^[1]唯一采用MongoDB的主从复制方法的情况是我们需要超过51个节点的时候，因为可复制集群无法拥有超过50个节点，当然这在通常情况下非常少见。

^[2]关于消费者硬盘驱动故障比率的深入分析可以阅读Google的文章《Failure Trends in a Large Disk Drive Population》(http://research.google.com/archive/disk_failures.pdf)。

重点注意的是，虽然可复制集群是冗余的，但是它也无法取代备份机制。备份是过去某个时间点上数据库数据的快照，可复制集群通常是最新的。有些时候数据集太大，可能导致备份不太符合实际。但是通常的规则下，备份是必须的，即使运行了复制机制也需要启动备份。换句话说，备份是为了预防逻辑故障，比如突发性数据丢失或者数据冲突。

我们强烈推荐在生产环境下启用复制和日志功能，除非你可以接受数据丢失。当然不推荐这种糟糕的部署方式。经历过系统故障失败后，就会发现思考和设置可复制机制需要花费额外的时间。

11.1.2 复制的使用场景和限制

你可能对于复制数据库的各种功能感到惊讶。特别是复制有利于冗余、故障切换、维护和负载均衡。我们来简要看看每个使用情况。

复制首先是为冗余设计的。它可以确保从节点与主节点的数据同步。这些复制节点可以和主节点位于一个数据中心或者为了安全可以分布于其他数据中心。复制是异步的，任何网络延迟和分区都不会影响主节点的性能。作为另外一种形式的冗余，复制节点也可以延迟某个固定的时间后执行。这防止了用户无意删除集合或者应用程序与数据库冲突的情况。通常，这些操作将会被立即复制；延迟复制可以给管理员足够的时间来做出响应并保存数据。

另外一个复制的使用情况是灾备。我们希望系统是高可用的，但是只有在冗余节点时才可以使用，并且在紧急情况下切换这些节点。MongoDB的可复制集群让这一切都可以自动切换，非常简单。

除了数据冗余和灾备，复制也简化了维护工作，通常通过允许管理员在从节点而不是主节点上运行命令。例如，通常的经验是在从节点上运行备份命令来缓解对于主节点的压力，避免宕机。构建大型索引是另外一个例子。因为所有构建非常昂贵，我们可以在从节点上优先构建，然后切换主从节点的角色，再次在新的从节点上构建索引。

最后，复制可以允许我们在从节点上平衡读/写压力。对于读取压力超大的应用系统，这个是最简单的解决办法，或者如果你选择最原始的做法，可以伸缩MongoDB数据库。但是对于所有的承诺，可复制集群在以下情况下无能为力：

- 现有的硬件无法处理工作负荷。例如，我们前面章节提到的工作集，如果数据集超过了可用的内存大小，那么随机读取从节点就不会像我们期望的那样改善性能。从可复制集群从节点读数据可以增加IOPS数量，但是100~200 IOPS可能无法解决性能问题，特别是同时写入并且消耗部分I/O数量的时候。此时，分片可能是最好的选择。
- 写入和读取的比例超过50%。这个比例是个好的开始点。由于每个主节点的写入都会最终写入每个从节点上，因此，直接读取已经在处理很多写入请求的从节点时会降低复制的性能，并且可能无法增加读写吞吐量。
- 应用程序需要一致性读取。从节点异步复制，因此不能确保反映最新主节点的写入数据。

在极端糟糕的情况下，从节点可能延迟几个小时。正如将会在后面介绍的，我们可以确保写数据请求在返回磁盘驱动器前到达从节点，但是这个方法有延迟开销。

对于不需要立即一致性系统，可复制集群是卓越的伸缩读并发的解决方案，但是并非适用于所有场景。如果需要伸缩系统，特殊的情况可能需要不同的策略，使用分片、升级硬件或者结合两个方案。

11.2 可复制集

Replica sets

可复制集是推荐使用的MongoDB复制策略。我们将从配置简单的可复制集开始，然后介绍复制的工作原理。这个知识对于分析生成环境的问题至关重要。最后讨论高级配置细节、备灾和恢复、最佳部署实践。

11.2.1 安装

可复制集推荐使用的最小配置包含3个节点，因为在可复制集里如果只有2个节点，那么一旦主节点垮掉，就无法进行多数投票表决。在3个成员的可复制集里可以包含3个存储数据的节点服务器，也可以是2个数据存储服务器加上1个裁判服务器。主节点是集群里唯一接受写入操作的服务器。可复制集群成员会通过一个叫做“选举”（elect）的过程来投票选出新成员。如果主节点不可用，选举机制可以在不需要人为参与的情况下选择新的主节点，允许正常的操作继续进行。不幸的是，如果可复制集中的大多数节点无法访问或者不可用，集群就无法接受写入并且所有的成员都会变成只读。如果在网络分区中的节点数据相同，就可以考虑加入一个裁判节点。此时，裁判会维护公平，保证集合正常选举出新的主节点。

在最小配置下，3个节点中的2个节点是第一级别，作为mongod实例。每个都可以作为可复制集的主节点，而且都有数据的完整拷贝。集合中的第3个节点作为裁判，它不需要复制数据，仅仅作作为观察者服务器。裁判是轻量级的mongod服务器，它只参与主节点选举，但是不会复制任何数据。可复制集如图11.1所示。裁判位于右边从数据中心。

现在我们来创建一个3个节点的可复制集来演示如何创建可复制集。通常，我们应该在独立的机器上分别创建每个节点。为了简化例子，我们会把3个MongoDB节点实例放在一个机器上。每个MongoDB实例通过主机名和端口区分；本地运行可复制集意味着，连接时我们会使用本地主机名在不同的端口上启动服务。

现在从创建每个节点的数据目录文件夹开始：

```
mkdir ~/node1
mkdir ~/node2
mkdir ~/arbiter
```

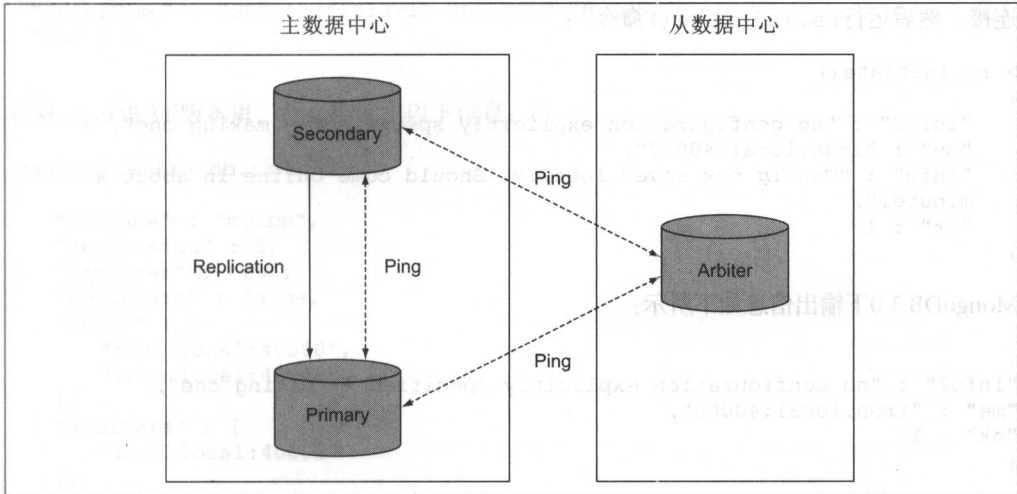



图 11.1 最基本的可复制集，包含一个主节点、一个从节点和一个裁判节点

接下来，启动每个mongod实例。由于我们会在同一台机器上运行每个进程，因此最简单的办法是在单独的终端命令窗口里运行各自的mongod命令：

```
mongod --replSet myapp --dbpath ~/node1 --port 40000
mongod --replSet myapp --dbpath ~/node2 --port 40001
mongod --replSet myapp --dbpath ~/arbiter --port 40002
```

注意，我们要启动的mongod实例属于myapp集群，并且在不同的端口上启动。如果要检查myapp的输出日志，首先注意到的错误信息是无法找到配置文件。

这很正常：

```
[rsStart] replSet info you may need to run replSetInitiate
-- rs.initiate() in the shell -- if that is not already done
[rsStart] replSet can't get local.system.replset config from self
or any seed (EMPTYCONFIG)
```

在MongoDB 3.0里，日志消息如下所示：

```
2015-09-15T16:27:21.088+0300 I REPL [initandlisten] Did not find local
replica set configuration document at startup; NoMatchingDocument Did not
find replica set configuration document in local.system.replset
```

要继续进行，就需要配置可复制集。这样做，首先要连接非裁判节点。这些事例没有运行在MongoDB默认端口上，连接服务可以在shell里使用下面的命令：

```
mongo --port 40000
```

这些例子是在本地运行mongod进程实例，所以会看到机器的名字iron经常弹出。要替换为自己的机器名字。

连接, 然后运行rs.initiate()命令^[1]:

```
> rs.initiate()
{
  "info2" : "no configuration explicitly specified -- making one",
  "me" : "iron.local:40000",
  "info" : "Config now saved locally. Should come online in about a
minute.",
  "ok" : 1
}
```

MongoDB 3.0下输出信息如下所示:

```
{
  "info2" : "no configuration explicitly specified -- making one",
  "me" : "iron.local:40000",
  "ok" : 1
}
```

等一会就可以拥有一个成员的可复制集啦。现在可以继续使用rs.add()来添加其他2个服务器节点:

```
> rs.add("iron.local:40001")
{ "ok" : 1 }
> rs.add("iron.local:40002", {arbiterOnly: true})
{ "ok" : 1 }
```

在MongoDB 3.0里, 可以使用下面的命令来添加裁判成员:

```
> rs.addArb("iron.local:40002")
{ "ok" : 1 }
```

注意, 对第2个节点, 我们制定了arbiterOnly参数来创建裁判节点。1分钟以内, 所有的节点应该都可以在线了。要获得可复制集群的简要信息, 可以运行db.isMaster()命令来查看:

```
> db.isMaster()
{
  "setName" : "myapp",
  "ismaster" : true,
  "secondary" : false,
  "hosts" : [
    "iron.local:40001",
    "iron.local:40000"
  ],
  "arbiters" : [
    "iron.local:40002"
  ],
  "primary" : "iron.local:40000",
  "me" : "iron.local:40000",
  "maxBsonObjectSize" : 16777216,
  "maxMessageSizeBytes" : 48000000,
}
```

^[1] 有些用户反映使用这个步骤时有问题, 因为配置文件包含 bind_ip=127.0.0.1 一行信息, mongod.conf 在/etc/mongod.conf or/usr/local/etc/mongod.conf 里。如果允许可复制集出错, 那么可以删除这个配置。

```

"localTime" : ISODate("2013-11-06T05:53:25.538Z"),
"ok" : 1
}

```

在MongoDB 3.0版本里，此命令输出以下信息：

```

myapp:PRIMARY> db.isMaster()
{
  "setName" : "myapp",
  "setVersion" : 5,
  "ismaster" : true,
  "secondary" : false,
  "hosts" : [
    "iron.local:40000",
    "iron.local:40001"
  ],
  "arbiters" : [
    "iron.local:40002"
  ],
  "primary" : "iron.local:40000",
  "me" : "iron.local:40000",
  "electionId" : ObjectId("55f81dd44a50a01e0e3b4ede"),
  "maxBsonObjectSize" : 16777216,
  "maxMessageSizeBytes" : 48000000,
  "maxWriteBatchSize" : 1000,
  "localTime" : ISODate("2015-09-15T13:37:13.798Z"),
  "maxWireVersion" : 3,
  "minWireVersion" : 0,
  "ok" : 1
}

```

更详细的信息可以通过rs.status()方法查看。这样可以查看每个节点信息。以下是完整的状态信息列表：

```

> rs.status()
{
  "set" : "myapp",
  "date" : ISODate("2013-11-07T17:01:29Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "iron.local:40000",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 1099,
      "optime" : Timestamp(1383842561, 1),
      "optimeDate" : ISODate("2013-11-07T16:42:41Z"),
      "self" : true
    },
    {
      "_id" : 1,
      "name" : "iron.local:40001",
      "health" : 1,
      "state" : 2,

```

```

"stateStr" : "SECONDARY",
"uptime" : 1091,
"optime" : Timestamp(1383842561, 1),
"optimeDate" : ISODate("2013-11-07T16:42:41Z"),
"lastHeartbeat" : ISODate("2013-11-07T17:01:29Z"),
"lastHeartbeatRecv" : ISODate("2013-11-07T17:01:29Z"),
"pingMs" : 0,
"lastHeartbeatMessage" : "syncing to: iron.local:40000",
"syncingTo" : "iron.local:40000"
},
{
  "_id" : 2,
  "name" : "iron.local:40002",
  "health" : 1,
  "state" : 7,
  "stateStr" : "ARBITER",
  "uptime" : 1089,
  "lastHeartbeat" : ISODate("2013-11-07T17:01:29Z"),
  "lastHeartbeatRecv" : ISODate("2013-11-07T17:01:29Z"),
  "pingMs" : 0
}
],
  "ok" : 1
}

```

在MongoDB 3.0版本里，rs.status()命令生成的结果信息则有所不同：

```

{
  "set" : "myapp",
  "date" : ISODate("2015-09-15T13:41:58.772Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "iron.local:40000",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 878,
      "optime" : Timestamp(1442324156, 1),
      "optimeDate" : ISODate("2015-09-15T13:35:56Z"),
      "electionTime" : Timestamp(1442323924, 2),
      "electionDate" : ISODate("2015-09-15T13:32:04Z"),
      "configVersion" : 5,
      "self" : true
    },
    {
      "_id" : 1,
      "name" : "iron.local:40001",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 473,
      "optime" : Timestamp(1442324156, 1),
      "optimeDate" : ISODate("2015-09-15T13:35:56Z"),
      "lastHeartbeat" : ISODate("2015-09-15T13:41:56.819Z"),
      "lastHeartbeatRecv" : ISODate("2015-09-15T13:41:57.396Z"),

```

```

    "pingMs" : 0,
    "syncingTo" : "iron.local:40000",
    "configVersion" : 5
  },
  {
    "_id" : 2,
    "name" : "iron.local:40002",
    "health" : 1,
    "state" : 7,
    "stateStr" : "ARBITER",
    "uptime" : 360,
    "lastHeartbeat" : ISODate("2015-09-15T13:41:57.676Z"),
    "lastHeartbeatRecv" : ISODate("2015-09-15T13:41:57.676Z"),
    "pingMs" : 10,
    "configVersion" : 5
  }
],
"ok" : 1
}

```

除非MongoDB包含许多数据，可复制集应该可以在30秒内上线。在这个时间里，每个节点的stateStr字段信息应该会从RECOVERING转换为PRIMARY、SECONDARY或者ARBITER。

虽然状态信息表明可复制集已经开始正常工作，但是我们可能希望获得更多的验证。现在通过使用shell连接主节点来插入一个文档数据进行测试：

```

$ mongo --port 40000
myapp:PRIMARY> use bookstore
switched to db bookstore
myapp:PRIMARY> db.books.insert({title: "Oliver Twist"})
myapp:PRIMARY> show dbs
bookstore 0.203125GB
local 0.203125GB

```

注意，MongoDB shell打印了它连接到的可复制集群中主节点的成员信息。

不出意外会立即发生数据复制。在另外一个终端窗口里，打开一个新的shell实例，但这次只连接从节点服务实例。查询刚刚插入的数据，结果如下所示：

```

$ mongo --port 40001
myapp:SECONDARY> show dbs
bookstore 0.203125GB
local 0.203125GB
myapp:SECONDARY> use bookstore
switched to db bookstore
myapp:SECONDARY> rs.slaveOk()
myapp:SECONDARY> db.books.find()
{ "_id" : ObjectId("4d42ebf28e3c0c32c06bdf20"), "title" : "Oliver Twist" }

```

如果复制工作如上所示，说明已经成功配置了可复制集。默认情况下，MongoDB会阻止对于从服务器的查询，因为数据可能比主节点慢。所以必须通过在从节点的shell里运行rs.slaveOk()命令来设置从节点允许读取操作。

看到复制的数据应该很满意。但是更有意思的可能是自动化灾备。我们现在来试验杀死一个节点。我们可以杀死一个从服务实例，但是它只会停止复制，其他服务器保持原来的状态。如果要查看系统的状态转换，就要杀死主节点服务器。如果主节点在shell的前台运行，则可以通过直接按下快捷键Ctrl+C来停止进程。如果在后台运行，则可以在~/node1的mongod.lock文件里获取进程ID，运行kill -3 <process id>命令即可。我们可以使用shell连接主服务器，使用下面的命令关闭主服务器进程：

```
$ mongo --port 40000
```

```
PRIMARY> use admin
```

```
PRIMARY> db.shutdownServer()
```

一旦杀死主服务器进程，从节点就会探测主节点心跳的信息。然后从节点会选举自己作为主节点。这个选举是可能的，因为主节点的大多数节点（裁判和最初的从节点）仍然可以ping通彼此。这是一些从节点的日志信息：

```
Thu Nov 7 09:23:23.091 [rsHealthPoll] replset info iron.local:40000
heartbeat failed, retrying
Thu Nov 7 09:23:23.091 [rsHealthPoll] replSet info iron.local:40000
is down (or slow to respond):
Thu Nov 7 09:23:23.091 [rsHealthPoll] replSet member iron.local:40000
is now in state DOWN
Thu Nov 7 09:23:23.092 [rsMgr] replSet info electSelf 1
Thu Nov 7 09:23:23.202 [rsMgr] replSet PRIMARY
```

如果你连接新的主节点并且检查可复制集的状态，就会看到旧的节点已经不可达了：

```
$ mongo --port 40001
```

```
> rs.status()
```

```
...
{
  "_id" : 0,
  "name" : "iron.local:40000",
  "health" : 0,
  "state" : 8,
  "stateStr" : "(not reachable/healthy)",
  "uptime" : 0,
  "optime" : Timestamp(1383844267, 1),
  "optimeDate" : ISODate("2013-11-07T17:11:07Z"),
  "lastHeartbeat" : ISODate("2013-11-07T17:30:00Z"),
  "lastHeartbeatRecv" : ISODate("2013-11-07T17:23:21Z"),
  "pingMs" : 0
},
:
```

后故障转移的可复制集，只有2个节点。因为裁判没有数据，应用程序就会继续工作，只要它可以正常连接主节点^[1]。虽然如此，但是没有发生复制，现在没有可能性进行灾备了。旧的主

^[1]有时候为了伸缩吞吐量，应用程序会从从节点读取数据。如果这样，这种故障就会导致无法读取数据。所以在脑子里设计好应用程序的灾备非常重要。本章结束部分会做更详细的介绍。

节点必须被恢复。假设旧的主节点完全关闭了，我们可以重新启动并让它上线，而且它会作为从节点重新加入到可复制集中。现在我们尝试重新启动旧的主节点。

这是对可复制集的快速预览。某些详细内容可能有点混乱。在下面两节里，我们会深入学习可复制集如何工作，以及它的部署和高级配置选项，还有如何处理生成环境下出现的比较棘手的问题。

11.2.2 可复制集群工作原理

可复制集依赖两个基本的机制：oplog和heartbeat。oplog允许复制数据，heartbeat监控状态并促发灾备。我们来看这两个机制如何协调工作。现在应该来理解并预测可复制集的行为，尤其是失败场景。

OPLOG 的全部

MongoDB的复制机制依赖oplog。oplog是个盖子集合，它存在于每个复制节点的local数据库里，而且记录了所有的数据变化。每次客户端写入主节点的数据，包含足够重生信息的项目就会添加到主节点的oplog中。一旦写入的数据被复制到某个从节点，从节点的oplog也会存储这个写入请求的记录。每个oplog项目通过BSON的时间戳来区分，并且所有的从节点使用时间戳来跟踪它们应用的最新项目^[1]。

为了更好地理解它的工作原理，我们来深入看一下真实的oplog，以及其中记录的操作信息。首先在shell里连接前一节里的主节点，然后切换到local数据库：

```
myapp:PRIMARY> use local
switched to db local
```

Local数据库存储了所有的可复制集群中的元数据和oplog操作日志。自然地，这个数据库不是复制的数据库本身。因此它有自己的名字，local数据库中的数据在本地节点被认为是唯一的，而且不能重复。

如果检查local数据库，就会看到名为oplog.rs的集合，这就是可复制集存储oplog的地方。我们也会看到一些其他的集合。以下是完整的输出信息：

```
myapp:PRIMARY> show collections
me
oplog.rs
replset.minvalid
slaves
startup_log
system.indexes
system.replset
```

^[1]BSON 时间戳是个唯一的标示符，它由从纪元开始的秒数数字组成，是个递增的计数器。更多信息可以查看http://en.wikipedia.org/wiki/Unix_time。

replset.minvalid包含可复制集成员最初同步的信息，system.replset存储了可复制集的配置文档。并非所有的mongod服务器都有replset.minvalid集合。me和slaves用来实现写关注点，这会在本章结束的部分介绍。system.indexes是标准的索引容器。

首先我们关注oplog。我们查询前一节里添加的与图书相关的oplog信息。为此，要输入下面的查询语句。结果文档有4个字段，我们将会依次讨论每一个字段：

```
> db.oplog.rs.findOne({op: "i"})
{
  "ts" : Timestamp(1383844267, 1),
  "h" : NumberLong("-305734463742602323"),
  "v" : 2,
  "op" : "i",
  "ns" : "bookstore.books",
  "o" : {
    "_id" : ObjectId("527bc9aac2595f18349e4154"),
    "title" : "Oliver Twist"
  }
}
```

第一个字段ts存储的是项目的BSON时间戳。时间戳包含2个部分：第一个部分表示从纪元（1970-01-01 00:00:00 UTC）开始的秒数，第二个部分表示计数器值——这里是1。使用时间戳查询，需要显示指定一个时间戳对象。所有的驱动都有自己的BSON时间戳构造函数，而且JavaScript也有。下面是如何使用它的例子：

```
db.oplog.rs.findOne({ts: Timestamp(1383844267, 1)})
```

回到oplog项目，op字段指定opcode操作代码。它可以告诉从节点oplog项目表示的操作。这里我们看到的是i，表示插入操作。Op后面是ns，它用来定义命名空间（数据库和集合），然后小写的字母o表示查询操作包含了一个查询文档的副本。

当我们查看oplog项目时，可能会注意到影响多个文档的操作被单独记录日志。对于多个更新或者大量的删除，每个受影响的文档都会单独创建oplog文档。例如，假如我们要在集合中插入更多的Dickens的书：

```
myapp:PRIMARY> use bookstore
myapp:PRIMARY> db.books.insert({title: "A Tale of Two Cities"})
myapp:PRIMARY> db.books.insert({title: "Great Expectations"})
```

现在集合中有4本书，我们来更新作者信息，添加作者的名字：

```
myapp:PRIMARY> db.books.update({}, {$set: {author: "Dickens {multi:true}}})
```

它是怎么出现在oplog中的呢？

```
myapp:PRIMARY> use local
myapp:PRIMARY> db.oplog.rs.find({op: "u"})
{
  "ts" : Timestamp(1384128758, 1),
  "h" : NumberLong("5431582342821118204"),
  "v" : 2,
```

```

"op" : "u",
"ns" : "bookstore.books",
"o2" : {
  "_id" : ObjectId("527bc9aac2595f18349e4154")
},
"o" : {
  "$set" : {
    "author" : "Dickens"
  }
}
}
{
  "ts" : Timestamp(1384128758, 2),
  "h" : NumberLong("3897436474689294423"),
  "v" : 2,
  "op" : "u",
  "ns" : "bookstore.books",
  "o2" : {
    "_id" : ObjectId("528020a9f3f61863aba207e7")
  },
  "o" : {
    "$set" : {
      "author" : "Dickens"
    }
  }
}
{
  "ts" : Timestamp(1384128758, 3),
  "h" : NumberLong("2241781384783113"),
  "v" : 2,
  "op" : "u",
  "ns" : "bookstore.books",
  "o2" : {
    "_id" : ObjectId("528020a9f3f61863aba207e8")
  },
  "o" : {
    "$set" : {
      "author" : "Dickens"
    }
  }
}

```

正如你看到的，每个更新的文档都会有自己的oplog记录。这个过程是更通用的主从复制策略的一部分，用来确保主从节点一直有相同的数据。要保证这一点，每个应用的操作必须是幂等的——它无法干涉应用了多少次oplog项目。但是结果必须相同。其他多文档的操作，比如删除，将会出现相同的行为。我们可以尝试不同的操作来看看oplog的最终记录。

要获取oplog当前状态信息，可以在shell里运行db.getReplicationInfo()方法获取信息：

```

myapp:PRIMARY> db.getReplicationInfo()
{
  "logSizeMB" : 192,
  "usedMB" : 0.01,
  "timeDiff" : 286197,
  "timeDiffHours" : 79.5,

```

```

    "tFirst" : "Thu Nov 07 2013 08:42:41 GMT-0800 (PST)",
    "tLast"  : "Sun Nov 10 2013 16:12:38 GMT-0800 (PST)",
    "now"    : "Sun Nov 10 2013 16:19:49 GMT-0800 (PST)"
  }
}

```

这里可以在oplog里看到第一个和最后一个的项目。可以通过使用\$natural排序修饰符来手动找到这些oplog记录。例如，下面是查询最新的项目：

```
db.oplog.rs.find().sort({$natural: -1}) .limit(1)
```

理解复制机制剩下的最后一点是，从节点追踪自己的oplog的步骤。答案是像从节点一样保存了oplog。这是对于主从复制的一大改进，所以值得花时间来研究底层实现。

想像一下，在主节点上写了个数据。接下来发生什么？一开始，写操作会被记录并添加到主节点的oplog里。同时，所有的从节点都会复制主节点的oplog。当某个从节点准备更新自己的数据时，它会做3件事情。首先，它会查看自己oplog里最新记录的时间戳。然后，它会查询主节点的oplog，查询所有时间戳大于自己当前时间戳的oplog记录。最后，它写入数据，并添加每个操作日志到自己的oplog里^[1]。这意味着，假如出现故障，任意的提升为主节点的从节点都会有一个oplog，这样其他的从节点可以复制。这个特性本质上支持了可复制集的故障恢复功能。

从节点使用了长轮训来立即应用从主节点复制的oplog记录。长轮训意味着从节点向主节点发送了个长请求。当主节点接受修改时，它会立即响应等待的请求。因此，从节点将几乎可以做到实时更新。当它们落后时，因为网络分区或者从节点维护，每个从节点里的oplog可以用来监控任意落后的复制。

主 从 复 制

主从复制是MongoDB最早使用的复制方式。这种复制易于配置，并且可以支持任意数量的从节点服务器。但是主从复制并非生产环境下推荐的方式。这有一些原因。首先，灾备都是完全人工的。

如果主节点发生故障失败，管理员必须关闭一个从服务器，然后作为主节点重新启动。然后应用程序必须重新配置连接新的主节点。其次，恢复非常困难。因为oplog只在主节点存在，故障失败需要新的服务器上创建新的oplog。这意味着任意存在的节点需要重新从新的主节点同步oplog。

简而言之，很难有有说服力的理由来使用主从复制。可复制集群是最佳的替代方案，而且这也是推荐的复制方式。如果说必须有一些原因来使用主从复制，则可以查询MongoDB手册获取更多信息。

^[1]当启用日志的时候，在同一个事务里，文档被写入核心数据文件，而且同时写入 oplog 。

停止复制

如果从节点无法在主节点oplog找到同步的日志记录点,就会永久停止复制。发生这种问题时,我们会看到从节点日志里有这样的异常信息:

```
repl: replication data too stale, halting
Fri Jan 28 14:19:27 [replsecondary] caught SyncException
```

回忆一下,oplog是个盖子集合。这意味着集合只能存储固定数量的数据。如果从节点离线一段时间,oplog可能无法存储这段时间里的所有改变记录。一旦从节点无法从主节点找到同步的oplog点,就无法确保从节点是主节点的完美复制集合了。

因为唯一的停止复制的补救措施就是完整地重新同步主节点的数据,肯定想避免这种问题。为此,需要去监控从节点的延迟,而且必须有足够大的oplog空间。我们会在第12章里介绍更多关于监控的知识。选择正确的oplog大小是我们下面要介绍的内容。

复制集 Oplog 的大小

oplog是个盖子集合,因此,MongoDB 2.6不允许在创建后再修改其大小。所以非常重要的一开始就仔细选择oplog的大小。在MongoDB 3.0里,我们可以修改oplog的大小。操作步骤就是,停止mongod实例,然后作为独立节点启动,修改oplog大小,重新启动成员。

默认的oplog大小都不一样。在32位系统上是50MB,在64位系统上是1GB或者5%的可用磁盘空间大小,除非正运行在Mac OS X上,此时它比较特殊,其oplog是192MB。这是因为我们假设OS X系统只能作为开发机器。对于部署来说,5%的可用磁盘空间已经足够了。一种考虑oplog大小的方法就是,一旦发现oplog被重写了20次,基本上就是表示磁盘满了(对于只有插入的服务器操作是真的)。

这里假设默认的大小无法满足实际应用的需求。如果知道自己的应用会有更高的写入量,就应该在部署前做一些实现性的测试。设置复制,然后以生产环境下的速度写入主节点。需要持续测试1个小时。一旦完成,可以连接任意可复制集成员,然后获取当前的复制信息。

```
db.getReplicationInfo()
```

一旦知道了每小时生成的oplog日志数量,就可以决定分配多大的oplog空间了。我们的目标是消除从节点与主节点oplog的差别,以保持实时更新。我们应该考虑从节点至少8个小时的离线时间。我们想避免完整的复制日志工作,并且增加oplog大小,这会在网络故障的情况下耗费更多的同步时间。

如果要修改默认的oplog大小,就必须先使用mongod -oplogSize参数启动每个节点。单位是兆字节。因此可以使用1GB的oplog空间启动mongod,如下所示^[1]:

```
mongod --replSet myapp --oplogSize 1024
```

^[1]对于如何创新调整 oplog 大小,请阅读 <http://docs.mongodb.org/manual/tutorial/change-oplog-size/>。

心跳和故障转移

可复制集心跳便于实现选举和灾备。默认情况下，每个可复制集委员会每隔2秒ping一次索引的其他成员。这样，系统就可以判断自己的健康状态。当运行`rs.status()`时，查看每个节点的最新心跳和状态（1表示健康，0表示无应答）。

只要每个节点仍然健康并且响应，可复制集就会继续正常工作。但是如果某个节点无法响应，就可能会采取行动了。每个可复制集都要确保一直只有一个主节点存在。这只是在大多数节点可用的时候。例如，回头看下前一节里建立的例子。如果关闭了某个从节点，但是大多数节点还存在，那么可复制集就不会改变，而是简单地等待从节点回来，重新上线。如果杀死了主节点，则大多数节点还存在，但是没有主节点，因此从节点会自动提升为主节点。如果多于一个从节点，则最新的从节点将会被选中。

但是也有其他可能的场景存在。想象下从节点和裁判都被杀死的情况。选中的主节点还在，但是没有其他节点——最初的节点只有一个是健康状态。此时，我们会在主节点的日志里看到如下的信息：

```
[rsMgr] can't see a majority of the set, relinquishing primary
[rsMgr] replSet relinquishing primary state
[rsMgr] replSet SECONDARY
[rsMgr] replSet closing client sockets after relinquishing primary
```

由于没有从节点，主节点降级为从节点。这看起来有点可笑，但是思考一下如果此节点依然是主节点会发生什么问题？如果是网络分区等原因导致心跳失败，则其他节点将会仍然在线。如果裁判和从节点仍然正常工作，并且可以看到彼此，则根据大多数原则，剩余的从节点会变成主节点。

如果主节点不退位，那么就会遇到一个尴尬的情况：一个可复制集有2个主节点。如果应用程序持续运行，它也许会向两台节点写入或者读取数据，肯定是不一致而且非常奇怪的行为。因此，当主节点看不到其他成员时，它必须退位降级。

提交和回滚

理解可复制集的最后重要知识点就是提交的概念。

从本质上讲，你可以一直向主节点写入数据，但是这些写入操作不会被提交，直到它们被复制给大多数从节点。那么什么是提交？这个概念最好通过例子来解答。

注意，对于MongoDB单个文档的操作都是原子性的，但是对于多个文档的操作并非原子性的。

想象一下前面一节里我们建立的可复制集群，假设我们在主节点插入了一些数据，但是由于某种原因没有及时复制到从节点（连接问题、从节点无法备份、从节点延迟等）。现在假设从节点突然提升为主节点，我们向新的主节点写入数据，但是最终旧的主节点又回来了，并且尝试从新的主节点复制数据。当旧的主节点中有一些写入数据在新主节点的oplog里不存在时，就会促发回滚。

在回滚里，所有没有复制给大多数节点的写入操作都会被取消。这意味着它们会被从节点oplog和集合里删除。如果从节点已经注册了个删除操作，则该节点将会从其他复制集里查询该文档并恢复它。对于文档更新和删除集合的操作恢复机制是一样的。

还原的写操作被存储在相关节点数据目录的子目录里。对于每个回滚写入的集合，都会创建一个单独的BSON文件，文件名包含回滚的时间。在需要恢复回滚文档的事件里，可以使用bsondump工具来检查BSON文件，并且手动恢复它们，可以使用mongorestore。

如果必须恢复回滚数据，就会发现这是我们想要极力去避免的。幸运的是，某种程度上也是可以的。如果应用程序可以容忍写延迟，那么可以使用写关注点，这个概念在后面介绍，它可以确保写数据被复制到大多数节点。聪明地选择写关注和监控复制延迟可以帮助我们减少回滚，或者避免它们。

本节里我们介绍了很多复制的内部机制，可能超过大家的预期，但是这些知识非常重要，大家应该记住。掌握复制的工作原理也许可以帮助我们诊断任意生产环境遇到的问题。

11.2.3 管理

对于所有的自动化功能，可复制集包含一些潜在的复杂配置参数。在下面的内容里，我们会详细介绍这些选项。基于大道至简的原则，我们也会说明哪些选项可以忽略。

配置细节

在这里我们会介绍mongod可复制集的启动选项，而且还将会介绍可复制集的配置文档。

复制选项

早期，我们学习了如何使用rs.initiate()和rs.add()在shell里启动可复制集。这些方法非常方便，但是它们隐藏了某些可复制集参数。我们来看一下如何使用配置文档来初始化并更新一个可复制集的配置信息。

配置文档指定了可复制集的配置信息。要创建配置文件，首先要为匹配replSet参数的_id添加一个值：

```
> config = { _id: "myapp", members: [] }
{ "_id" : "myapp", "members" : [ ] }
```

单个的members可以作为配置文件的一部分进行定义，如下所示：

```
config.members.push({ _id: 0, host: 'iron.local:40000' })
config.members.push({ _id: 1, host: 'iron.local:40001' })
config.members.push({ _id: 2, host: 'iron.local:40002', arbiterOnly: true })
```

正如前面强调的，iron是我们的机器名，替换了我们自己的主机名。现在配置文档应该如下所示：


```
> config
{
  "_id" : "myapp",
  "members" : [
    {
      "_id" : 0,
      "host" : "iron.local:40000"
    },
    {
      "_id" : 1,
      "host" : "iron.local:40001"
    },
    {
      "_id" : 2,
      "host" : "iron.local:40002",
      "arbiterOnly" : true
    }
  ]
}
```

可以通过把文档作为第一个参数传递给`rs.initiate()`来初始化可复制集合。

从技术上来说，此配置文档包含一个`_id`，它存储的是可复制集的名字，一个指定3~50个成员的数组，一个指定全局设置的子文档。这个例子中可复制集使用了最小的必备参数，加上可选的`arbiterOnly`设置。请记住，虽然一个可复制集有50个成员，但是它只有7个投票成员。

这个文档需要一个`_id`来匹配可复制集的名字。初始命令将会检验每个使用`replSet`启动的成员节点。每个可复制集`member`需要一个`_id`，这个`_id`由从0开始的自增整数组成。另外，`members`成员需要一个`host`字段存储主机名和可选端口字段。

假设使用`rs.initiate()`方法初始化了可复制集。`replSetInitiate`是个简单的封装命令。因此可用如下方式来启动可复制集：

```
db.runCommand({replSetInitiate: config});
```

`config`变量存储了配置文档。一旦初始化，每个集合成员都存储了这个配置的副本到本地的`local`数据的`system.replset`集合里。如果查询此集合，将会看到此文档有个版本号。无论何时修改可复制集配置，就必须再加此版本号。访问当前配置最简单的方式就是运行`rs.conf()`命令。

要修改可复制配置，还有一个单独的命令：`replSetReconfig`，它可以接受一个新的配置文档作为参数。或者我们也可以使用`rs.reconfig()`，它内部也使用了`replSetReconfig`命令。新的文档可以指定额外的或者删除的集合成员，并且一起修改成员和全局配置。修改配置文档的过程，增加了版本号，并且作为`replSetReconfig`的一部分传递，比较麻烦，所以许多`shell`帮助方法可以简化这些过程。要查看全部帮助方法，可以在`shell`里运行`rs.help()`。

记住，每当重新配置可复制集，都会导致选举新的主节点，所有的客户端都会关闭。这样可

以确保客户端不会写数据到从节点上。

如果你对使用客户端驱动来配置可复制集群感兴趣，可以阅读rs.add()的源码。在shell里输入rs.add(不要带括号)。下面来看一下此方法如何工作。

配置文档参数

直到现在，我们已经学习过了最简单的可复制集配置文档的编写工作上。但是，这些文档为可复制集以及成员提供了几个参数。我们从成员参数开始，已经看了_id、host、arbiterOnly。现在还有一些剩余的参数选项，详细介绍如下：

- **_id (必备)**——唯一的自增整数，用来表示成员ID。这个_id的值从0开始，而且必须是递增的。
- **host (必备)**——存储主机的名字和端口。如果提供了端口，就应该用冒号分割（例如iron:30000）。如果没有端口，默认端口是27017。我们在前面看到过，但是前面例子里使用的是简单的设置_id和host：

```
{
  "_id" : 0,
  "host" : "iron:40000"
}
```

- **arbiterOnly**——一个布尔值，true或者false，表示这个成员是否是裁判。裁判只存储配置信息。它们是轻量级的成员，只参与主节点选举工作，不参与复制数据。

下面是arbiterOnly的设置例子：

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "arbiterOnly": true
}
```

- **priority**——其数值指定此节点被选举为主节点的优先级。对于可复制集，开始和故障时都会尝试选举新的主节点，只要它的操作日志最新，就使用最高的优先级。这个参数非常有用，如果可复制集的某些节点比其他节点强大，则这个非常有用：它可以帮助简化选举出最强的机器成为主节点。

有时候可能我们喜欢某个节点永远不会成为主节点（例如，二级数据中心里的灾难恢复节点）。这种情况下，把服务器优先级设置为0。优先级为0的节点永远不会被选举成为主节点。下面是设置的例子：

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "priority" : 500
}
```

- **votes**——默认情况下所有的可复制集成员都有一个投票机会。votes可设置为允许给某个成员投多票。

这个参数使用的情况很少。因为，当不是所有的成员都有相同的票数时，就很难推测出可复制集的故障原因。此外，一人一票制可以满足各种不同的生产环境部署需求。如果要修改某个成员的投票数，就三思而后行，提前仔细模拟分析各种不同的故障场景。下面的成员有2票：

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "votes" : 2
}
```

- **hidden**——布尔值，为true时，用isMaster命令查询时不会出现应答消息。因为MongoDB驱动通过isMaster来确定可复制集的拓扑图，隐藏成员可以保持驱动访问此成员。这个设置可以与buildIndexes一起使用，而且必须与slaveDelay一起使用。此成员被配置为隐藏模式：

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "hidden" : true
}
```

- **buildIndexes**——布尔值，默认是true，用来决定此成员是否构建索引。只有当此成员从来不会变成主节点时才会设置为false（优先级priority为0）。

此参数是专为备份节点设计的。如果备份索引非常重要，就不应该使用此参数。下面是配置不构建索引的例子：

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "buildIndexes" : false
}
```

- **slaveDelay**——从节点与主节点复制数据的延迟时间。此参数只能用在从来不会成为主节点的机器上。要指定slaveDelay大于0，就要确保设置它的优先级为0。我们也可以使用特定时间的延迟来防范用户错误。例如，如果有30分钟的从服务器延迟，且管理员意外删除数据库，就可以有30分钟来处理该事件。下面的成员已经配置slaveDelay为1小时：

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "slaveDelay" : 3600
}
```

- **tags**——包含键值对的集合文档，通常用来区分某个数据中心的成员或服务器机房位置。标签用来指定写关注点的粒度和读设置，在11.3.4节会详细介绍。在tag文档里，输入的值必须是字符串。以下是有两个标签的成员配置文档例子：

```
{
  "_id" : 0,
  "host" : "iron:40000",
  "tags" : {
    "datacenter" : "NY",

```

```
"rack" : "B"
}
```

这些包含了单个可复制集成员的配置参数。还有2个全局的可复制集配置参数包含在 settings 键下。在可复制集配置文档里，它们看起来如下所示：

```
{
  _id: "myapp",
  members: [ ... ],
  settings: {
    getLastErrorDefaults: {
      w: 1
    },
    getLastErrorModes: {
      multiDC: {
        dc: 2
      }
    }
  }
}
```

- **getLastErrorDefaults**——当客户端无参数调用 `getLastError` 时使用一个指定默认参数的文档。这个选项应该仔细使用，因为它也可以在驱动里设置全局的默认 `getLastError`。可以想象一下，开发者调用 `getLastError`，却不知道管理员已经在服务器上指定了默认值。

关于 `getLastError` 的更多信息可以参考文档：<http://docs.mongodb.org/manual/reference/command/getLastError>。简单地说，要为所有写入操作都指定在500ms内复制到至少2个成员上，就可以像下面这样来指定配置文件的值：

```
settings: {
  getLastErrorDefaults: {
    w: 2,
    wtimeout: 500
  }
}
```

- **getLastErrorModes**——为 `getLastError` 命令定义了额外模式的文档。这个功能依赖于可复制集的标签，在11.3.4小节里做了详细介绍。

可复制集状态

可以通过运行 `replSetGetStatus` 命令来查看可复制集和成员的状态。在 shell 里调用此命令，运行 `rs.status()` 帮助方法，则结果文档将显示各个成员的状态、运行时间和日期操作次数。了解可复制集的成员状态信息非常重要。我们可以在表11.1里查看可能的状态值的完整列表。

当所有的节点在位移状态1、2、7，并且至少有一个节点是主节点时，可以认为可复制集稳定并且在线。也可以从外部脚本里使用 `rs.status()` 或者 `replSetGetStatus` 命令来监控集群

的整体状态、可复制延迟时间、运行时间，而且这也是生产部署推荐的方式^[1]。

表 11.1 可复制集状态

状态	状态字符串	备注
0	STARTUP	表示可复制集正在通过 ping 与其他节点成员写书并共享配置数据
1	PRIMARY	这是主节点。可复制集永远只有一个主节点
2	SECONDARY	这是个从节点，为只读节点。这个节点如果它的优先级大于 0,并且没有被标记为隐藏（hidden）模式，则可能在故障的时候变成主节点
3	RECOVERING	此节点现在无法读/写。通常在故障转移或者添加新节点后看到此状态。当恢复时，数据文件正在同步；可以通过同步查看节点的日志文件来验证它
4	FATAL	网络仍然在链接，但是节点对 ping 没有响应。通常这表示该节点出现了致命错误，被标记为 FATAL
5	STARTUP2	同步的初始状态
6	UNKNOWN	还没有建立网络连接
7	ARBITER	此节点是裁判
8	DOWN	节点可以访问，而且某些点上稳定的，但是当前不响应心跳 ping 消息
9	ROLLBACK	正在回滚
10	REMOVED	该节点曾经是可复制集的成员，但是现在已经删除了

故障转移和恢复

在可复制集例子里，我们看到了一些故障转移的例子。这里我们总结了一些故障转移的规则，并且提供了一些恢复的建议。

当集群中配置文件指定的所有成员都可以互相通信时，可复制集将会成功恢复回来。每个节点默认都有一票，而且这些投票会形成多数派，并选举主节点。这意味着可复制集最少有2个节点（2票）启动。但是最初的票数也可以决定在故障失败事件里组成怎样的多数派。

我们假设已经配置了最小的可复制集，包含3个节点（没有裁判），因此有推荐的自动化灾备最小数量服务器。如果主节点失败，并且其余的从节点可以互相联系通信，那么就可以选举出新的主节点。对于哪个是新的主节点，从节点会使用最新的操作日志oplog和最高的优先级来决定。

故障模式与恢复

恢复是在故障后恢复可复制集到最初状态的过程。

^[1]注意，除了运行状态命令，也可以通过 Web 控制台获取可视化结果。第 13 章讨论 Web 控制台并且展示了使用可复制集的例子。

总体上有2个失败类型要处理。第一个叫做干净失败。此时该节点的数据文件仍然是完整的。此种故障的例子之一的就是网络分区。如果一个节点与集群中的其他节点失去连接，就只能等待连接恢复，而且分区节点会重新成为集合中的一员。一个相似的情况发生在某个节点的mongod进程中中断时，但是能够重新干净地上线^[1]。一旦进程重新启动，它就可以重新加入集群中。

第二种类型叫做绝对失败。此时节点的数据文件要么丢失，要么冲突。不干净地关闭了mongod进程，而且还没有启用日志。磁盘驱动器故障就是此种失败的例子。绝对失败的唯一恢复手段就是通过再同步或者从最新的备份里恢复数据文件。我们依次来看看这两个策略。

要完全再同步，可以在故障节点上使用空的文件夹来重新启动mongod实例。只要主机和端口没有改变，新的mongod就会重新加入可复制集，然后重新同步现有的数据。如果主机或者端口改变，那么在mongod启动后，就要必须重新配置可复制集。例如，假设节点是iron:40001，它无法恢复了，我们就在foobar:40000重新启动mongod。可以通过抓取配置文档来重新配置可复制集，修改第二个节点的主机，然后传递给rs.reconfig()方法：

```
> config = rs.conf()
{
  "_id" : "myapp",
  "version" : 1,
  "members" : [
    {
      "_id" : 0,
      "host" : "iron:40000"
    },
    {
      "_id" : 1,
      "host" : "iron:40001"
    },
    {
      "_id" : 2,
      "host" : "iron:40002",
      "arbiterOnly" : true
    }
  ]
}
> config.members[1].host = "foobar:40000"
foobar:40000
> rs.reconfig(config)
```

现在可复制集将会识别新的节点，并且新的节点应该开始从现有的成员同步数据了。

通过完整再同步数据来恢复节点，我们还可以从最近的备份中恢复数据。可以通过从一个从节点获取数据文件的快照来离线存储备份^[2]。通过备份恢复只有当备份文件的日期与当前集合中的成员的操作日志日期相差不太久时才行。这意味着备份中的oplog必须在当前的操作日志

^[1]如果 MongoDB 干净地关闭，就表明它的数据文件是良好的。如果运行日志分析，MongoDB 应该可以恢复，无论它是如何停止的。

^[2]第 13 章会详细讨论备份。

中。我们可以使用`db.getReplicationInfo()`提供的信息来检查是否如此。当操作的时候，不要忘记恢复数据需要的实际。如果备份的最新操作日志与复制新机器的实际差距太久时，建议还是执行完整的再同步操作。

从备份恢复的速度比较快，因为不需要重建索引。要从备份恢复数据，就复制备份的数据文件到`mongod`数据路径下。然后它会自动开始再同步，而且我们可以通过检查日志或者运行`rs.status()`来检验这个过程。

部署策略

我们知道，在MongoDB 3.0里的可复制集可以包含50个节点，而且我们也学习过了许多配置参数，思考过故障转移和恢复了。许多配置可复制集群的方式，但是在本节里我们只会介绍大多数情况下适用的方法。

最小可复制集配置提供了自动化灾备故障转移功能，我们之前建立的集合包含2个节点和一个裁判。生产环境下，裁判可以运行在应用服务器上，而其他节点可以有自己的服务器。这个配置比较省钱，而且对于大多数生产环境的App已经够用了。

对于应用程序，正常运行的时间至关重要，我们希望可复制集只由3个复制节点组成。额外的复制节点做什么？考虑一下单个节点完全故障失败的情况。当我们恢复第3个节点时还有2个一级节点可用。只要第3个节点正常上线，并且恢复数据（可能花费几小时），在此期间可复制集仍然可以自动故障转移。

某些应用可能需要两个数据中心来提供冗余，3个成员的可复制集也可以正常完成工作。方法就是使用一个数据中心用于灾难恢复。图11.2展示了这个例子。这里的主数据中心机房托管了一个主节点和一个从节点，备份数据中心保证另外一个从节点，作为被动节点（优先级为0）。

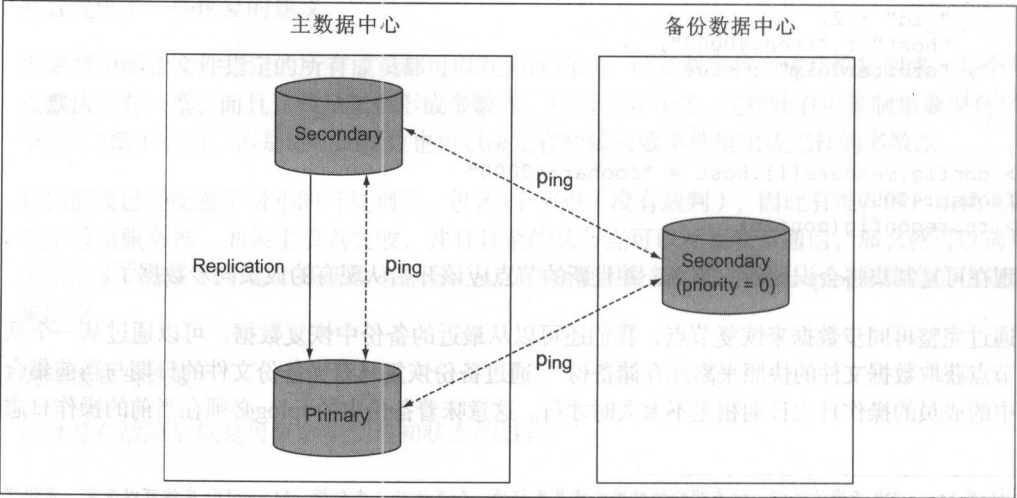


图 11.2 分布于两个数据中心的 3 个节点的可复制集

在这个配置里，可复制集主节点将会一直运行在A数据中心。可能会丢失1个节点或者任意的数据中心，但是可以保持数据应用在线。故障转移通常是自动的，除非A中的2个节点都丢失。因为不太可能一次丢失2个节点，否则表示完全故障或者A数据中心故障。要快速恢复，我们可以关闭数据中心B的成员，然后不使用`--replSet`重新启动它。还有一个办法，我们可以在数据中心B里启动2个新节点，然后强制可复制集重新配置。当集群中的大多数节点都无法联系时，不建议重新配置可复制集，但是可以在紧急情况下使用`force`参数这样做。例如，如果已经定义了新的配置文件`config`，可以使用下面的命令强制重新启动：

```
> rs.reconfig(config, {force: true})
```

对于生产环境，测试非常关键。确保发布到生产环境之前测试所有典型的故障转移和恢复场景。实战经历过这些故障失败的场景后，对于后期发生的问题就会更加镇定自若，解决故障并恢复可复制集也会信心倍增。

11.3 驱动与复制

Drivers and replication

如果使用MongoDB的复制机制，就需要知道几个应用相关的主题。首先就是连接与故障转移。然后就是写关注点，它允许我们决定在应用继续之前什么程度下的写数据应该被复制。接下来的主题是读伸缩，允许应用在复制集群里多台机器上分散读请求压力。最后，我们将会讨论标签，一种配置更复杂可复制集群读/写的方式。

11.3.1 连接与故障转移

MongoDB驱动为连接可复制集群提供了统一的接口。

单节点连接

连接可复制集群的主节点与连接之前例子中的普通MongoDB节点没有区别。两种情况下，驱动都会初始化TCP socket连接，然后运行`isMaster`命令。对于单独的节点，这个命令返回如下信息：

```
{
  "ismaster" : true,
  "maxBsonObjectSize" : 16777216,
  "maxMessageSizeBytes" : 48000000,
  "localTime" : ISODate("2013-11-12T05:22:54.317Z"),
  "ok" : 1
}
```

对于驱动来说，最重要的是`isMaster`自动被设置为`true`，表示该节点是独立的节点，或者

是主从复制或可复制集中的主节点^[1]。在所有这些情况中，此节点可以被写入，也可以使用驱动执行任意的CRUD操作。

当直接连接可复制集的从节点时，我们必须知道我们正在连接的节点（对于绝大多数驱动至少是这样的）。在Ruby驱动里，可以使用`:read`参数指定节点。要直接连接第一个从节点，Ruby代码如下所示：

```
@con = Mongo::Client.new(['iron: 40001'], {:read => {:mode => :secondary}})
```

若没有`:read`参数，驱动就会抛出异常，表示它无法连接主节点（假设mongod从节点运行端口是40001）。这个检查可以避免无意从从节点读取数据。虽然这种尝试新读取通常都会被服务器拒绝，但不会看到任何异常，除非在启动安全模式下运行此操作。

这种假设是大家通常想连接主节点，`:read`参数强制作智能检测。

可复制集连接

我们也可以单独连接可复制集里的单个节点，但是通常我们想作为整体来连接整个可复制集。这需要驱动知道哪个是主节点，并且在故障出现的时候，重新连接新的主节点。

绝大部分官方的驱动提供了连接可复制集的方式。在Ruby里，我们可以通过创建新的`Mongo::Client`实例、传递服务器节点列表和可复制集的名字来进行连接操作：

```
Mongo::Client.new(['iron:40000', 'iron:40001'], :replica_set => 'myapp')
```

从内部来说，驱动将会尝试连接每个节点，然后调用`isMaster`命令。此命令会返回许多重要的可复制集的详细信息：

```
> db.isMaster()
{
  "setName" : "myapp",
  "ismaster" : false,
  "secondary" : true,
  "hosts" : [
    "iron:40001",
    "iron:40000"
  ],
  "arbiters" : [
    "iron:40002"
  ],
  "me" : "iron:40000",
  "maxBsonObjectSize" : 16777216,
  "maxMessageSizeBytes" : 48000000,
  "localTime" : ISODate("2013-11-12T05:14:42.009Z"),
  "ok" : 1
}
```

^[1] `isMaster` 命令也返回了此版本服务器允许的最大 BSON 对象的大小。驱动可以在插入对象之前验证所有的 BSON 对象符合限制要求。

一旦节点应答返回这些信息，驱动就获得了所有需要的东西。

现在它可以连接主节点了，再次验证此节点是主节点，然后允许用户通过此节点进行读/写操作。应答对象也允许驱动来缓存剩下的从节点和裁判的地址信息。如果主节点上的操作失败，驱动的后续请求就可以连接其余的节点直到它可以重新连接主节点。

当使用这种方式连接MongoDB可复制集时，驱动会自动查找其他节点。这意味着当我们连接可复制集时，不需要列举每个成员。`isMaster`命令的返回消息会用来更新其他成员的信息。如果列举在连接参数里的成员没有活动的，连接会失败，所以，可以尽可能多地列举节点地址。但是不用担心，如果在连接列表里少写了某些节点，它们会自动找到。如果有多个数据中心，最好包含所有数据中心的成员。

要记住重要的一点，虽然可复制集的故障转移是自动的，但驱动不会隐藏故障发生的事实。处理的过程如下：主节点故障或者选举新的主节点。后续的请求会发现socket连接已经破坏了，驱动会抛出一个连接异常并关闭任意数据库的socket连接。接下来是应用程序开发者决定接下来的处理工作了，这个决定依赖于执行的操作和应用的特定需求。记住，驱动会自动重新连接并尝试发送请求。我们可以假想一些场景。假设我们只从数据库读取数据。此时，尝试读取失败的机会很少，因为几乎没有修改数据库状态。但是现在假设我们也要写入数据库。我们写入数据库的时候可以包含或者不包含错误检查。正如11.3.2小节讨论的，使写关注点为1或者更大，驱动就会检查问题，包括通过调用`getLastError`命令，进行可复制集的写入失败检查。这是MongoDB 2.0之后绝大多数驱动默认的方式，但是如果显示设置写关注点为0，驱动就不会检查错误写入数据到TCP socket连接。如果使用的是相对新的驱动版本或者shell，就没有必要显示调用`getLastError()`；任意情况下，写请求都会发生详细的ACK。

如果应用程序使用的写关注点值为0，并且发生了故障转移，就会出现不确定的状态。最近对数据库服务器发送了多少写请求？有多少丢失在socket缓存里？TCP socket写入数据的不定性让回答这个问题变得实际上不可能。问题有多严重取决于应用本身。对于日志，非安全模式的写入可能是可以接受的，因为丢失写入请求几乎没有改变日志的总体格局。但是对于创建数据的用户来说，无安全模式的写入可能是个灾难。

重要的是要记住，默认情况下写关注点设置为1，意味着写入请求会确定到达可复制集的某个成员上，而且设置为0存在风险。接受应答不会降低回滚的可能性，正如我们在11.2.2节里讨论的。MongoDB给了我们一些更高级的功能来管理这个问题，可以通过写关注点来控制如何写入。

11.3.2 写关注点

现在应该非常清楚，默认的写关注点值为1，是合理的。对于某些应用程序，知道写请求安全到达主节点非常重要。如果要减少回滚，更高级别确保是必须的，而且写关注点通过允许开发者在收到答复之前指定写请求应该被复制的程度，并且允许应用程序继续执行。从技术上

讲，通过`getLastError`命令上的两个参数控制写关注点：`w`和`wtimeout`。第一个参数`w`表示最新的写请求应该被复制到的服务器数量；第二个参数是如果写请求无法复制到其他节点的超时毫秒数，超过此时间就会返回错误。

例如，如果想要确保某个写请求被复制到至少一台机器上，可以设置`w`的值为2；如果想要在500 ms内完成操作，就需要设置`wtimeout`为500。注意，如果不指定`wtimeout`的值，并且由于某些原因没有复制，操作就会无限期阻塞。

当使用驱动时，通常要在写文档里传递写关注点值，但是它依赖于具体的驱动API。在Ruby里，可以在单个的操作上指定写关注，如下所示：

```
@collection.insert_one(doc, {w => 2, wtimeout => 200})
```

许多驱动支持为某个连接或者数据库设置默认的写关注点值。也可以为单个操作设置，如上所示，但是会成为整个连接的默认值：

```
Mongo::Client.new(['hostname:27017'], :write => {w => 2})
```

也有更炫的参数存在。例如，如果要启用日志，也可以通过添加`j`参数强制日志在写请求返回确认消息之前同步到磁盘。

```
@collection.insert_one(doc, :write => {w => 2, j => true})
```

要知道如何在特别的情况下设置写关注点，可以查询专门的驱动文档。

写关注点可以与可复制集合主从复制一起使用。如果检查`local`数据库，就会看到一些集合，`me`在从节点上，`slaves`在主节点上。它们用来实现写关注点。无论何时从节点轮训主节点，主节点会在`slaves`集合里记录每个从节点的最新操作日志`oplog`信息。因此主节点知道哪个节点在一直复制数据，因此可以响应`getLastError`命令的请求。

记住，使用`w`值大于1的写关注点会带来额外的延迟。可配置的写关注点本质上允许我们在速度和持久化之间做出平衡。如果再启用日志，则`w`的值为1时基本可以满足绝大多数应用的需求。换句话说，对于日志和分析，应该一起禁用日志和写关注点，只依赖复制来提供持久性。如果这样，则可能会在失败的时候丢失某些写请求。仔细考虑这些选择，并且在设计应用的时候仔细考虑。

11.3.3 读伸缩

对于读伸缩来说，复制数据库是很好的解决方案。如果单个服务器无法处理应用的读压力，就可以把查询请求路由到可复制集中的多台服务器上。绝大部分驱动已经内置了根据读取配置文件来向从节点发送查询请求的功能。使用Ruby驱动，这个可以作为`Mongo::Client`构造函数的一个参数进行设置：


```
Mongo::Client.new(
  ['iron:40000', 'iron:40001'],
  {:read => {:mode => :secondary}})
```

注意，在连接代码里，我们配置了新客户端读取的服务器地址。当`:read`参数被设置为`{:mode => :secondary}`时，连接对象会随机选择，从附近的从节点读取数据。这个配置叫做读偏好(read preference)，而且可以用来告诉驱动从特定的节点读取数据。绝大部分MongoDB驱动支持读偏好设置。

- *primary*——这是默认的设置，表明只从可复制集主节点读取数据，因此一直是连续的。如果可复制集有问题，并且没有可用的从节点，就表示出现错误。
- *primaryPreferred*——设置了此参数的驱动会从主节点读取数据，除非某些原因使主节点不可用或者没有主节点，此时它会从从节点读取数据。这意味着读请求无法保证一致性。
- *secondary*——这个设置告诉驱动应该一直从从节点读取数据。这种设置对于我们想确保读请求不会影响主节点的写入请求时非常有用。如果没有可用的从节点，读请求会抛出异常。
- *secondaryPreferred*——相比前面的设置，这个更加灵活。读请求会发送到从节点，除非没有从节点可用，此时会从主节点读取。
- *nearest*——此配置的驱动会尝试从最近的可复制集成员节点读取数据，通过网络延迟判断。可以是主节点也可以是从节点。因此读请求只会发送给驱动认为最快通信的节点。

记住，主节点读偏好是唯一一个可以确保读一致的模式。写请求首先在主节点完成。因为更新从服务器有些延迟，所以可能在从节点无法找到刚刚在主节点写入的文档数据，除非我们正在读取主节点服务器。

它证明即使不用`nearest`设置，如果MongoDB驱动已经设置了查询从节点的读偏好，那么它仍然会尝试去与最近的节点通信。这样做的根据是它的成员选举策略。驱动首先会根据网络延迟来排名所有的节点，然后会排除网络延迟大于15 ms的服务器，最后会随机选举剩余的节点之一。15 ms是一些驱动可以接受的网络延迟值。

对于Ruby驱动来说，配置如下所示：

```
Mongo::Client.new(
  ['iron:40000', 'iron:40001'],
  :read => {:mode => :secondary}, :local_threshold => '0.0015')
```

`local_threshold`参数指定浮点类型的最大网络延迟时间。

注意，`nearest`读偏好使用这个策略从节点中选择一个来读取数据，但是它在选择过程中也包括了主节点。整体上讲，这个方法的优势就是比完全选随机选举所有的节点有更低的网络延迟时间，但是如果它们有相似的延迟时间，仍然可以分发请求到多个节点。

许多MongoDB开发者在生产环境里使用复制来伸缩系统，但是也有三种情况下使用这种伸缩方式略显不足。第一种情况，与需要的服务器数量有关。MongoDB 2.0时，可复制集支持最大12个节点，7个可以投票。对于MongoDB 3.0，可复制集支持最大50个节点，7个可以投票。

如果需要更多的节点进行伸缩，可以使用主从模式。但是如果不牺牲自动化灾备和故障转移机制而又需要超过最大可复制集限制，就需要迁移到分片集群（sharded cluster）中。

第二种情况，应用带有高写入压力。正如本章开始的时候提到的，从节点必须跟上写请求的节奏。发送读请求给写满负载的从节点可能影响复制功能。

第三种情况，复制伸缩无法解决读一致性的问题。由于复制是异步的，复制无法反映主节点最新的写入数据，因此，如果应用任意从节点读取数据，就无法保证最终用户看到的数据是一致性的。对于某些应用，这不是问题，但是对于其他来说就需要一致性读。在我们第4章的购物车例子里，如果我们读取的不是最新的数据，就会有严重的问题。事实上，原子操作的读和写都必须在主节点完成。这些情况下，我们有2个选择。其中一个就是分离一致性读和不需要一致性读的操作。

前者可以一直从主节点读取，后者可以分散到从节点。当测试要么是太复杂或者不伸缩时，分片就是唯一的方法^[1]。

11.3.4 标签

如果我们正在使用写关注点或者读伸缩，就可能会发现自己想要对于从哪个节点读/写数据做更细粒度的控制。例如，假设我们有5个节点部署在两个数据中心里，这两个数据中心的地理位置是分开的，分别是NY和FR。主数据中心NY包含3个节点。从数据中心FR包含2个节点。我们假设要使用写关注点来阻塞，直到写请求被复制到至少一个在FR数据中心的服务器上。根据目前已经知道的关于写关注点的知识，我们会看到没有好的解决办法，无法在大多数节点上设置w的值为3，因为最可能的场景是3个在NY的节点会最先收到确认消息。也可以使用4，但是这个值当每个数据中心丢失了一个节点时就没有帮助了。

可复制集标签通过允许为可复制集中的成员设置特定的标签并使用标签定义写关注解决了这个问题。要了解它如何工作，首先需要学习如何在可复制集成员上设置标签。在配置文档里，每个成员都有一个tags的键指向一个包含键值对的对象。

这是个例子：

```
{
  "_id" : "myapp",
  "version" : 1,
  "members" : [
    {
      "_id" : 0,
      "host" : "ny1.myapp.com:30000",
      "tags": { "dc": "NY", "rackNY": "A" }
    },
    {
      "_id" : 1,
```

^[1]注意，从分片集群里获取读一致性，我们必须一直从每个分片的主节点上读取数据，并且使用安全写模式。

```

"host" : "ny2.myapp.com:30000",
"tags": { "dc": "NY", "rackNY": "A" }
},
{
  "_id" : 2,
  "host" : "ny3.myapp.com:30000",
  "tags": { "dc": "NY", "rackNY": "B" }
},
{
  "_id" : 3,
  "host" : "fr1.myapp.com:30000",
  "tags": { "dc": "FR", "rackFR": "A" }
},
{
  "_id" : 4,
  "host" : "fr2.myapp.com:30000",
  "tags": { "dc": "FR", "rackFR": "B" }
}
],
settings: {
  getLastErrorModes: {
    multiDC: { dc: 2 } },
    multiRack: { rackNY: 2 } },
  }
}

```

这是跨越两个数据中心的假想的可复制集，标签配置文档。注意，每个成员的标签文档保护2个键值对：第一个表示数据中心，第二个表示服务器机柜。记住，这里使用的名字是完全随意的，而且只在本文例子有意义。我们也可以在标签文档里设置任意内容（值必须是字符串）。最重要的是如何使用它。

这也是getLastErrorModes产生的原因。它允许我们为getLastError命令定义不同的模式，它实行了特定的写关注点需求。在这个例子中，我们定义了2个模式。第一个是multiDC，定义如{ "dc": 2 }，它表示写请求应该被复制给dc，标记了至少两个不同值的节点。如果检查标签，会看到这是确保写请求传播到2个数据中心必备的设置。第2个节点指定了NY的至少2个服务器机柜应该可以收到写请求。标签让这个更加清晰。

通常，getLastErrorModes项目至少包含一个键（此时是dc和rackNY），值是整数。这些整数表示正确执行getLastError命令时必须满足的不同标签的最少数量。一旦定义了这个模式，就可以在应用中使用它们作为w的值。例如，在Ruby使用第一个模式，代码如下：

```
@collection.with(:write => {w => "multiDC"}).insert_one(doc)
```

除了让写关注点更加复杂，标签也可以和11.3.3小节里介绍的读偏好机制一起使用。它可以通过标签限制读请求发送的服务器。例如，如果使用从节点读偏好，驱动会忽略不包含特定标签的从节点服务器。因为主节点读偏好只能从一个节点读取，所以它无法与标签兼容，但是索引洽谈的读偏好可以兼容。这是Ruby驱动使用读偏好标签的例子：

```
@collection.find({user: "pbakkum"},
{
  :read => :secondary,
  :tag_sets => {
    :dc => "NY"
  }
})
```

这个配置会从带有dc:NY标签的从节点上读取数据。

标签是MongoDB上一个你可能永远不会使用的元素，但是它在某些场景下可能非常有用。如果你在管理复杂的可复制集配置，那么请记住它。

11.4 总结

Summary

从我们所介绍的内容可以看出，对于部署来说复制机制必不可少。MongoDB的复制机制比较简单，而且配置也一样。但是对于备份和故障转移来说，有一些隐藏的复杂性。对于复制的情况，经验和本章的实战例子可以提供帮助。

在本章结束的时候，以下是在继续学习和管理自己的可复制集时要记住的关键知识点：

- 在MongoDB的生产环境部署时，数据保护应该使用可复制集。如果做不到这一点，就必须经常备份数据。
- 可复制集应该保护至少3个成员，其中一个可以是裁判。
- 数据不会提交，直到它被写入可复制集的大数据节点中。在故障的场景下，如果大多数成员存在，它们就继续接受写请求。没有达到大多数成员的写请求会被放入回滚数据目录下，并且必须手动处理。
- 如果可复制集从节点宕机一段时间，并且数据库修改没有记录到MongoDB的oplog里，这个节点就没有办法及时填补数据，而且必须重新同步数据。为了避免这个问题，可以尝试最小化从节点的失败时间。
- 驱动的写关注点控制着在返回之前必须有多少个节点被写入。增加这个值可增加持久性。对于正式的持久性，我们推荐设置为大多数成员以避免回滚场景，但这个方法会带来延迟成本。
- MongoDB允许我们通过读偏好和标签来控制更复杂的可复制集的读/写行为，特别是在多个数据中心部署可复制集成员时。

一如既往，我们要仔细考虑部署和测试。当有效使用可复制集时，它会成为无价的工具。

使用分片集群扩展系统

Scaling your system with sharding

本章内容

- 分片的动机和架构
- 设置与加载一个例子分片集群
- 查询并建立分片集群索引
- 选择分片集群的键
- 在生产环境下部署分片集群

随着应用系统规模的增长，成本会变得越来越贵，而且有时候无法实现使用单台机器来处理负载压力。这种问题的一个解决方案就是汇聚大量低价、低处理能力的机器来解决问题。MongoDB的分片(sharding)就是为解决这种问题而设计的：把超大数据使用更小的片进行分区存储，这样就不需要在单个机器上存储所有的数据或者承担全部压力。MongoDB分片对于应用系统是透明的，这意味着对于分片集群的查询与可复制或者单个mongod服务器实例的查询完全一样。

本章会从分片集群的总体介绍开始，深入介绍分片集群要解决的问题，以及如何知道什么时候需要它。接下来，我们将会讨论组成分片集群的组件。最后，我们会介绍两种不同的分片方式，以及解开MongoDB基于范围的分片秘密。

前三节内容会给大家最基本的关于分片集群工作的知识与概念，只有当我们自己配置完分片集群后才会完全掌握这些概念如何一起工作。这也是我们要在第4节里要完成的工作。我们会构建一个例子集群来托管类Google Docs应用程序的大量数据。

我们会讨论一些分片机制，描述查询和索引如何跨片工作。我们也会介绍日益重要的分片集群key的选择策略，本章最后我们会总结一些生产环境下运行分片集群必备的建议。

Google 文档替代电商案例

本章我们会使用类似Google文档的应用来替代之前的电商应用例子，因为它的文档定义更加简单，并且允许我们关注分片本身。

在电商应用中，可能有多个集合，比如存储用户评论的集合；有的可能更小，比如存储用户配置信息的集合。在更复杂的应用中，可以为了获得更大容量而分片存储超大型集合数据；为了保留较小的集合而不分片存储。分片和不分片的集合可以共存在一个系统里，可以一起工作，对于应用来说完全透明。事实上，如果后面发现不需要分片存储的集合变大，则也可以随时再启用分片存储机制。

在我们的类Google Docs应用中，可以看到它也使用了相同的原则。我们会坚持简化例子并专注于本章的新知识。

12.1 分片集群概述

Sharding overview

在开始构建第一个分片集群之前，最好先来了解一下分片集群背后的概念。在本节里，我们将会介绍分片集群解决的问题，讨论分片面对的挑战，然后介绍什么时候使用分片集群才是正确的解决方案。

12.1.1 什么是分片集群

分片是把大型数据集进行分区成更小的可管理的片的过程。到目前为止本书的所有内容都是把MongoDB作为一个单独的服务器，而且每个mongod都包含了完整的应用程序数据的拷贝。即使当使用可复制集时（第11章介绍的），每个复制节点也能完整复制其他节点的数据。对于大部分应用系统来说，在每个服务器上完成保存全部的数据是可以接受的。但是随着数据的增长，应用系统需要更大的读/写吞吐量，单台服务器可能无法满足需求。

特别是，这些服务器可能没有足够的内存或者CPU核心处理庞大的负载压力。此外，随着数据的增长，在单个磁盘或者RAID磁盘阵列上存储并管理如此大的数据集也不太实际。如果要继续使用单台服务器或者虚拟硬件来托管数据库，则解决方案是在多台服务器上分散存储这些数据。此方法在MongoDB里叫做分片。MongoDB中的分片可以帮助应用程序进行扩展，但是记住，它是个大锤子。一个十分复杂的系统，增加了管理和性能的开销，所以在实施前请确定这是应用系统需要的。在接下来的小节里我们将会介绍如何鉴别分片是不是系统最佳的选择。

分片集群:实践出真知

分片集群比较复杂。要完全掌握本章的知识，需要实战运行例子代码。在单个机器上运行整个例子集群完全没有问题，一旦成功配置完机器，就可以开始试验。没有比分片部署集群更好的学习MongoDB分布式架构的方式了。

12.1.2 什么时候分片?

什么时候分片的问题在理论上非常简单,但是必须完全理解系统是怎么使用的。通常有两种分片情况:存储分布式和负载分布式。记住,分片不能解决所有的性能问题,而且它会增加额外的复杂性和开销,所以理解为什么要用分片非常重要。在许多情况下,分片集群可能不是最佳的解决方案。

存储分布式

通常理解系统的存储需求并不困难。MongoDB会把所有数据存储在最开始dbpath参数指定的路径里,这可以在附录A里获取更多的信息,所以,我们应该可以使用系统里提供的实用工具来监控 MongoDB 的使用情况。此外,在mongo shell里运行db.stats()和db.collection.stats()目录,会分别输出当前数据库和集合的存储使用信息。

如果仔细监控数据库的存储容量,随着应用的增长,就可以清晰地知道什么时候存储需求会超过任意一个节点的容量。

负载分布式

要理解负载,必须首先明确客户端使用的CPU、内存和I/O带宽,系统支持的必须比需求的略大。在第8章里,我们讨论了在内存中维护索引和工作数据集的重要性,其实这也是最常见的分片原因。如果应用的数据集持续无限增长,肯定会有个时间点出现应用的数据无法加载到内存里。如果在Amazon的EC2上运行,当超过最大可用内存时就会触发门槛。还有,虽然你可能在自己的硬件上运行MongoDB,自己可以增加内存,这样可能延迟分片的时间,但是没有机器有无限内存,因此,最终必须要做分片集群。

要确定服务器和处理负载量以及可用内存的关系,通常不是这么简单。例如,使用SSD磁盘或者磁盘阵列配置可以增加IOPS(磁盘处理的每秒的输入/输出操作),它可以将数据存入内存而不降低性能。也可能是工作的数据集只是全部数据的一小部分,这样就只需要相对小的内存。另外,如果在写负载方面有特殊的要求,最好进行良好的分片存储设计,比如想在数据达到内存极限之前进行分片操作,因为这样可以通过跨机器分散负载压力来获取更高的写吞吐量。

预 留 空 间

虽然可以等到磁盘100%用完并且机器过载的时候再进行分片,但是这是个糟糕的主意。分片本身会对系统本身产生一些额外的压力,因为自动化负载均衡的过程必须从过载的机器上转移压力到别的机器。如果系统已经过载,而没有发生负载均衡,则空的分片还是空的,过载的分片还是过载的,系统就会出现无法响应的状况。第13章就如何跟踪一些重要的指标给出了一些实际的建议,所以我们可以随着应用系统的增长平滑地扩张系统。

无论什么情况,对现有系统进行分片肯定要基于网络使用、磁盘使用、CPU使用、工作集日

益重要的比例、正常使用数据的数量和可用的内存大小。

既然已经掌握了分片集群的背景和理论知识，并且知道什么时候需要它，那么现在就来了解一下在MongoDB分片集群中需要使用的组件。

12.2 理解分片集群的组件

Understanding components of a sharded cluster

几个组件协同工作才可以让分片集群正常工作。当它们都正常工作时，这个整体就叫做分片集群。要理解MongoDB分片集群的工作原理，就需要了解所有组成分片集群的组件以及每个组件在集群中的角色。

分片集群由分片、mongos路由器、配置服务器组成，如图12.1所示。

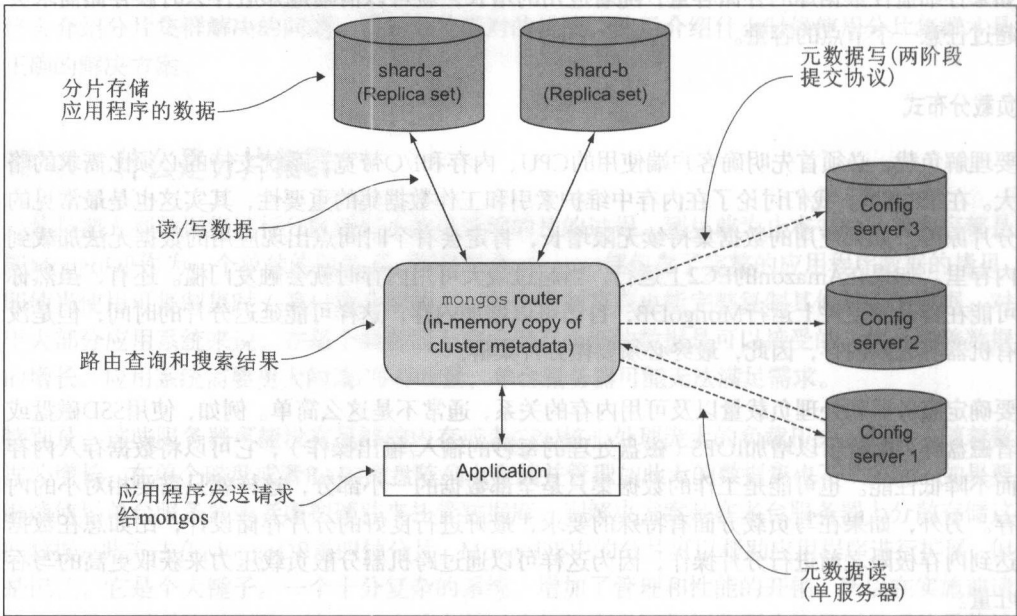


图 12.1 MongoDB 分片集群中的组件

我们来检查一下图12.1中的每个组件。

- 分片(左上)：存储了应用程序的数据。在分片集群中，只有mongos路由器或者系统管理员可以直接连接分片服务器节点。与不分片的部署一样，每个分片可以单独作为开发和测试的节点，但是生产环境下应该是个可复制集。
- mongos路由器（中心）：缓存了集群的元数据并使用它来路由操作到正确的分片服务器。
- 配置服务器（右上）：一直存储集群的元数据，包括哪个分片包含哪些数据集。

现在，我们来讨论每个组件角色的更多细节，以及它们在集群中的作用。

12.2.1 分片：存储应用程序数据

分片，如图12.1左上角所示的，或者是单个mongod服务器或者是一个存储部分应用数据的可复制集。事实上，分片是分片集群中应用程序数据存储的唯一位置。用于测试，一个片可以是单个的mongod服务器，但是在生产环境中应该是可复制集，因为它会有自己的复制机制并且可以自动灾备和故障转移。可以连接到一个片，也可以连接到单个服务器节点或者可复制集。但是如果尝试在这个片上直接运行操作，则只能看到整个集群数据中的一部分。

12.2.2 mongos 路由：路由操作

因为每个分片包含的只是总数据的一部分，所以需要某个东西来路由操作到适当的片上。这就是mongos的作用所在。mongos进程如图12.1的中心部分所示，是个可以直接转发所有读、写命令到正确的分片上的路由器。mongos提供给客户端单点连接集群的方式，这使得整个集群看起来就像单个节点一样。

mongos进程是轻量级的、不持久化的^[1]。因此，它们通常部署在应用程序服务器上，以确保只有一个网络跃点来转发请求。换句话说，应用程序连接本地的mongos，而mongos管理与每个分片服务器连接。

12.2.3 配置服务器：存储元数据

mongos进程是非持久化的，这意味着必须有某个东西来存储集群的元数据。这个工作就由配置服务器来做，如图12.1右上角所示。这些元数据包含全局的集群配置信息，每个数据库、集合、特定数据的范围的位置，以及保存了跨片数据迁移历史的一个修改日志。

配置服务器保存的元数据是集群正常工作和更新维护的关键。例如，每次mongos启动，mongos都会从配置服务器获取一份元数据拷贝。没有这些数据，就没有办法完整预览整个集群。这些元数据的重要性体现在通过配置服务器来保存分片集群设计和部署的策略数据。

图12.1所示中有3台配置服务器，但不是可复制集。它们需要比异步复制更强大的东西；当mongos进程写入它们时，使用的是两阶段提交协议^[2]。

这样可以确保跨配置服务器的一致性。我们必须在分片集群生产环境下部署3台配置服务器，

^[1] mongos 服务器在本地内存里缓存了配置服务器的元数据。这些元数据有个版本标识，当元数据修改的时候它也会改变，所以当带有旧数据的 mongos 尝试去连接新版本元数据的分片时，它就会收到一个更新本地元数据拷贝的提醒。

^[2] 【译者注】两阶段提交协议在分布式事务中体现得比较直观，我们在高级架构班讲解分布式事务的原理时解释过。需要分布式事务协调器和资源管理器介入。

而且这些服务器必须部署在单独的冗余机器上^[1]。

现在虽然知道了分片集群的组成与架构，但是可能仍然对分片集群本身好奇。数据实际上是如何分布式存储的？我们将会在下节里解释，首先介绍MongoDB里的两种分片方式，然后再介绍核心分片操作。

12.3 在分片集群中分散数据

Distributing data in a sharded cluster

在讨论分片的不同方式之前，我们来讨论一下在MongoDB中数据如何分组以及如何组织数据。

要解释这个问题，我们来讨论一下本章后面会建立的类Google Docs的应用程序。图12.2显示了这种应用在MongoDB中的数据结构。

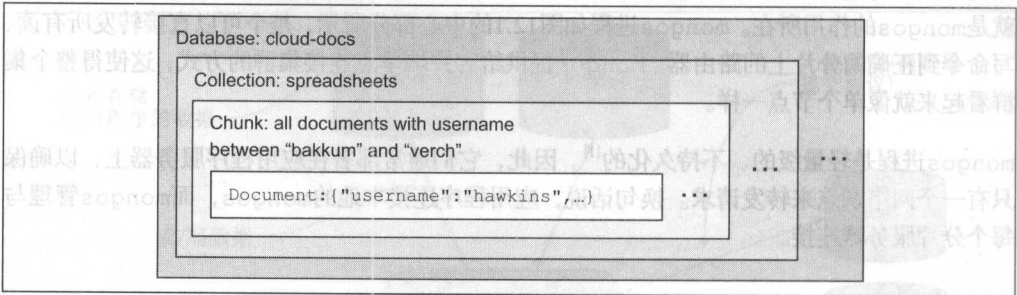


图 12.2 MongoDB 分片集群中可用的粒度级别

从最里面依次向外看，可以看到MongoDB有4种不同的分片级别：文档、块、集合和数据库。这4种不同的粒度级别代表着MongoDB中不同的数据单位。

- 文档：MongoDB中最小的数据单元。文档表示系统中不能够再分了的单个对象。我们可以把它与关系型数据库中的行相比较。注意，我们把文档及其字段作为一个原子性单元。在图12.2的最里层，可以看到使用了username字段，它的值为"hawkins"。
- 块：根据某个字段的值集群的一组文档。块是只在集群组里存在的概念。这是根据一个或者一组字段的值进行逻辑分组，这些字段称为分片键。本节在深入学习块的时候会介绍分片键，12.6节里会再次介绍。如图12.2中，块就是所有username的值在"bakkum"和"verch"之间的文档。
- 集合：数据库中一组命名的文档。允许用户把数据库分组为有意义的逻辑组，MongoDB提供了集合概念。这就是一组命名的文档，而且它必须被应用显示指定才能进行查询。图12.2所示中，集合名字叫spreadsheets。

^[1] 也可以运行一个配置服务器，但只有一个简单的测试分片集群的方法。生产环境只运行一个配置服务器就好比在大西洋上乘坐一个单引擎的飞机：它也许可以成功抵达目的地，但是也可能会让乘客见上帝。

集合的名字表明了它就是clouddocs数据库一个组的数据，我们会在后面讨论。

- 数据库：包含文档的集合。这是系统中最顶级的命名组。因为数据库包含文档的集合，所以集合也必须指定在文档上执行的操作。在图12.2所示中，数据库名字是cloud-docs。要运行查询，集合必须指定为—spreadsheets。组合数据库名字和集合的名字提供了系统中的唯一性，这通常也叫做命名空间（namespace）。它通常是集合名字和数据库名字连接在一起，用圆点分割。例如，图12.2所示的命名空间组合为cloud-docs.spreadsheets。

数据库和集合在4.3节里做过介绍，而且是无分片模式部署的，所以这里不熟悉的概念应该是块。

12.3.1 分片集群中的数据分散方式

现在我们已经知道了MongoDB中数据的逻辑组织方式，接下来的问题是，如何与分片集群交互？在哪个片上分区数据？问题的快捷答案就是数据可以分布到集群中4个分组中的2个上：

- 在整个数据库级别 这里每个数据库的所有集合都存在于自己的分片上。
- 分区或者集合分块级别 这里集合里的文档被分散到多个片上，根据某个或者多个字段字（分片键）进行分片。

你可能好奇，为什么MongoDB分区是基于块而不是单个的文档的？它看起来好像是最逻辑化的分组方式，因为文档是最小的可能数据单位。但是事实上，不仅要分区数据，而且还要能够找到它们，你会看到如果要在文档级别分区——例如，允许每个类Google Docs应用中的spreadsheet表格独立被移动——就要在配置服务器存储元数据，单独追踪每个文档。如果一个包含小文档的系统，则一半的数据可能是元数据，存储在配置服务器上，用于追踪实际的数据存储位置。

粒度从数据库跳到集合分区上

你也许好奇，为什么粒度会从数据库级别跳到集合分级别？为什么没有中间步骤我们可以在整个集合级别来分布式存储数据，而不是在集合级别分区？

这个问题的真实答案是它是理论上完全可行的，但实际没有实现。幸运的是，依靠数据库和集合之间的关系，有个简单的解决办法。如果你现在遇到的问题是不同的集合——假设是files.spreadsheets和files.powerpoints——需要放置在不同的服务器上，则可以把它们存储在不同的数据库里。例如，可以在files_spreadsheets.spreadsheets存储电子表格文件，而在files_powerpoints.powerpoints存储幻灯片文件。因为files_spreadsheets和files_powerpoints是不同的数据库，而且是分布式的，所以集合也是分布式的。

下面两节里，我们将会介绍每种分布式方法。首先我们会讨论分布式整个数据库，然后我们

会继续介绍常见的分布式集合块的方法。

12.3.2 分布式数据库分片

当我们在分片集群中创建数据库时，每个数据库分配到不同的片上。如果什么都不做，此数据库和它的所有集合都会永久存储在创建的片服务器上。这个数据库本身甚至不需要分片。

因为数据库的名字是通过应用来指定的，所以可以把它作为一种人工分区方式。MongoDB与分区毫无关系。想要知道为什么是人工方式，就思考使用这个方法来自分片spreadsheets集合。可以使用数据库分布式方法进行分片，但必须把数据库分为2个——假设叫files1和files2——最终在files1.spreadsheets和files2.spreadsheets中存储数据。完全由你来决定表格数据存储到哪个集合里，以及如何在后续的查询中找到正确的数据库。

这个问题比较困难，这也是为什么我们不推荐此方法的原因。

那么数据库分布式方法什么时候比较有用呢？一个比较真实的分布式数据库例子就是把MongoDB作为服务器。在此模型的一个实现中，客户可以付钱访问单个MongoDB数据库。在后台，每个数据库创建在单独的分片集群中。这意味着，如果每个客户使用相同数量的数据，则数据的分布式将是最佳的，因为数据库分布于整个集群中。

12.3.3 集合分片

现在，我们来看看MongoDB分片集群更强大的形式：分片单个集合。这也是自动分片一词的本意，因为这是MongoDB做出分区决定的分片形式，不需要应用参与。

为分片单个集合，MongoDB定义了块的概念，它是一个基于预定义字段值或分片键的逻辑分组文档。选择分片键是用户的职责，而且我们将会在12.8节中介绍。

例如，思考下面选自电子表格管理应用数据库的文档例子：

```
{
  _id: ObjectId("4d6e9b89b600c2c196442c21")
  filename: "spreadsheet-1",
  updated_at: ISODate("2011-03-02T19:22:54.845Z"),
  username: "banks",
  data: "raw document data"
}
```

如果集合中的所有文档都包含这种格式，就可以选择_id和username字段作为分片键。然后MongoDB就会使用每个文档中的这些信息来决定文档存储到哪个块中。

MongoDB是怎么做出决定的呢？在底层核心中，MongoDB的分片是基于范围的，这意味着每个块表示一个范围的键值。当MongoDB查看某个文档以确定它属于哪个块时，首先它会抽取分片键的值，然后找出包含分片键所在的块。

看一下具体的例子，假设我们已经为spreadsheets集合选择了username作为分片键，而且我们有2个片，“A”和“B”。我们的块分布可能有点类似于表12.1。

表 12.1 块和片

开始	结束	片
-∞	Abbot	B
Abbot	Dayton	A
Dayton	Harris	B
Harris	Norris	A
Norris	∞	B

从上表就会明白分片集群中块的作用了。如果我们有个文档的username值为"Babbage"，则看一下上面的表就知道它应该属于片A服务器。事实上，如果我们给大家任意包含username字段的文档，此字段是分片集群的键，你就可以使用表12.1确定此文档所属的数据块，然后决定它应该被发送到哪个分片服务器。我们将会在12.5和12.6两节里深入介绍这两个过程。

12.4 构建一个例子分片集群

Building a sample shard cluster

掌握分片最好的方式就是看看它实际怎么工作。幸运的是，可以在单台机器上创建一个分片集群，这也正是我们现在要做的^[1]。

设置分片集群的完整过程包含下面三个步骤。

(1) 启动mongod和mongos服务器——启动组成分片集群的单个进程mongod和mongos。在本章配置的集群里，我们将会启动9台mongod服务器和1台mongos服务器。

(2) 配置集群——更新配置信息，这样可复制集就可以初始化并且添加分片到集群里。在此之后，节点就可以互相通信。

(3) 分片集群——分片一个集合，这样它就可以被分散到多个片上。这个步骤单独分开的原因是，MongoDB在同一个集群里存在分片和不分片的集合，所以必须明确告诉集群要分片的集合。本章里，我们将会只分片一个集合，它就是cloud-docs数据库中的spreadsheets集合。

在接下来的三节内容里，我们会详细介绍这三步，然后模拟前一节里描述的基于云计算的电子表格应用例子行为。贯穿本章中，我们将会介绍全局分片的配置，在最后一节里，我们还会使用它来看看数据如何基于分片键来分区数据。

^[1] 我们可以在单个机器上运行每个 mongod 和 mongos 进程作为测试用。在 12.7 节里我们将会看看生产环境希望的分片配置，还有部署需要的最少机器的数量。

12.4.1 启动 mongod 和 mongos 服务器

创建分片集群的第一步就是启动所有必需的mongod和mongos进程。我们要建立的分片集群包含2个分片和3个配置服务器。我们也会启动一个单独的mongos来与集群进行通信。图12.3展示了所有启动的进程关系，括号里是使用的端口。

我们会使用命令来配置分片集群，所以如果无法看清命令参数，则可以回头来参考下面的集群图片。

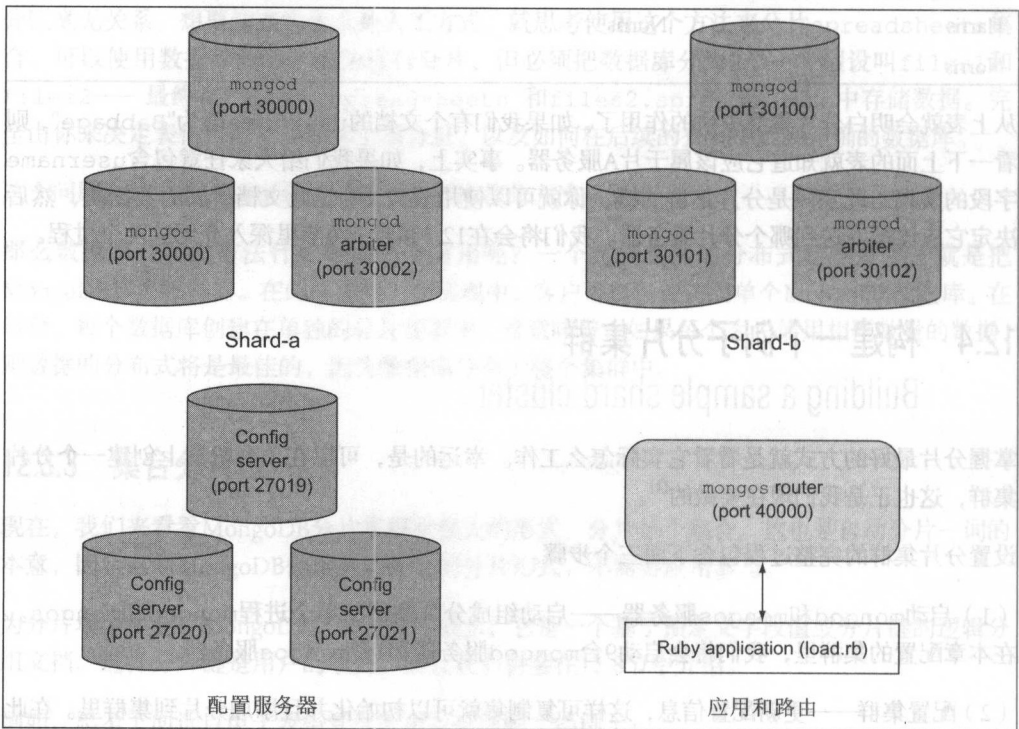


图 12.3 例子分片集群的进程关系图

启动分片集群组件

我们来创建几个文件夹作为两个可复制集的数据目录，这两个可复制集会成为集群的2个片节点：

```
$ mkdir /data/rs-a-1
$ mkdir /data/rs-a-2
$ mkdir /data/rs-a-3
$ mkdir /data/rs-b-1
$ mkdir /data/rs-b-2
$ mkdir /data/rs-b-3
```

接下来,启动所有mongod。因为运行了很多进程,所以可以使用`-fork`参数在后台运行它们^[1]。启动第一个可复制集的命令如下所示:

```
$ mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-1 \
--port 30000 --logpath /data/rs-a-1.log --fork
$ mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-2 \
--port 30001 --logpath /data/rs-a-2.log --fork
$ mongod --shardsvr --replSet shard-a --dbpath /data/rs-a-3 \
--port 30002 --logpath /data/rs-a-3.log --fork
```

这是第二个可复制集群的配置命令:

```
$ mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-1 \
--port 30100 --logpath /data/rs-b-1.log --fork
$ mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-2 \
--port 30101 --logpath /data/rs-b-2.log --fork
$ mongod --shardsvr --replSet shard-b --dbpath /data/rs-b-3 \
--port 30102 --logpath /data/rs-b-3.log --fork
```

注意节点之间不同的选项

这里没有介绍所有的命令行参数。要查看每个参数的具体信息,最好参考MongoDB的mongod文档: <http://docs.mongodb.org/manual/reference/program/mongod/>。通常来说,需要初始化这些可复制集。分别连接每个服务器,并运行`rs.initiate()`,然后添加其余的节点。第一个命令如下所示:

```
$ mongo localhost:30000
> rs.initiate()
```

在初始化节点变成主节点之前要等几秒钟。在这个过程中,命令窗口会显示信息节点从`shard-a:SECONDARY`>提升到`shard-a:PRIMARY`。使用`rs.status()`命令也可以揭示背后的秘密。后面可以继续添加其他的节点到此集群中:

```
> rs.add("localhost:30001")
> rs.addArb("localhost:30002")
```

addArb 意味着添加此节点到可复制集

使用localhost后面可能会有问题,因为它只能适用于所有节点都在单个机器的情况。如果知道主机名,就可以用来避免这个问题。在Mac上,主界面应该看起来像MacBook-Pro.local。如果不知道主机名,就确保都使用localhost!

配置可复制集作为分片节点与配置单独使用的可复制集是一样的,所以如果不熟悉这个配置过程,则可以回头复习第10章的内容。

初始化可复制集的过程相似。再来一次,运行`rs.initiate()`等待一会:

```
$ mongo localhost:30100
> rs.initiate()
> rs.add("localhost:30100")
> rs.addArb("localhost:30101")
```

^[1]如果在Windows系统上运行,则`fork`不起作用。必须为每个进程重新启动一个终端,当然最好省略`logpath`日志路径。

最后, 通过在shell里运行`rs.status()`来检查两个可复制集。如果都正常运行, 接下来就可以启动配置服务器了^[1]。现在来创建每个配置服务器的数据目录, 然后使用`configsvr`参数来启动`mongod`进程:

```
$ mkdir /data/config-1
$ mongod --configsvr --dbpath /data/config-1 --port 27019 \
  --logpath /data/config-1.log --fork --nojournal
$ mkdir /data/config-2
$ mongod --configsvr --dbpath /data/config-2 --port 27020 \
  --logpath /data/config-2.log --fork --nojournal
$ mkdir /data/config-3
$ mongod --configsvr --dbpath /data/config-3 --port 27021 \
  --logpath /data/config-3.log --fork --nojournal
```

确保每个配置服务器都运行起来并且可以使用shell连接, 或者通过检查日志文件(`tail -f <log_file_path>`)来检验每个进程都侦听在配置的端口上。查看每个配置服务器的日志文件可以看到如下类似的信息:

```
Wed Mar 2 15:43:28 [initandlisten] waiting for connections on port 27020
Wed Mar 2 15:43:28 [websvr] web admin interface listening on port 28020
```

如果每个配置服务器已经运行, 就可以继续启动`mongos`进程。`mongos`必须使用`configdb`参数启动, 它包含一个逗号分隔的配置服务器地址^[2]:

```
$ mongos --configdb localhost:27019,localhost:27020,localhost:27021 \
  --logpath /data/mongos.log --fork --port 40000
```

再强调一次, 我们这里不会介绍所有的命令参数。如果想知道每个参数的意义, 请参考`mongos`的官方文档: <http://docs.mongodb.org/manual/reference/program/mongos/>。

12.4.2 配置集群

现在我们已经启动了图12.2所示的集群所需的所有`mongod`和`mongos`进程, 是时候来配置集群了。从连接`mongos`开始。为了简化任务, 我们可以使用分片集群帮助方法。这些方法属于全局对象`sh`。要查看所有的帮助方法可以运行`sh.help()`。

我们可以使用`addShard`命令来输入一系列配置命令。

此命令的帮助方法是`sh.addShard()`。这个方法接受一个可复制集名字的字符串, 跟着的是每个可复制节点的地址。这里我们指定了2个可复制集成员的地址, 但是不包含裁判的地址:

```
$ mongo localhost:40000
> sh.addShard("shard-a/localhost:30000,localhost:30001")
{ "shardAdded" : "shard-a", "ok" : 1 }
> sh.addShard("shard-b/localhost:30100,localhost:30101")
{ "shardAdded" : "shard-b", "ok" : 1 }
```

^[1]再强调一次, 如果使用 Windows 平台, 可以忽略`-fork`和`-logpath`参数, 在新命令窗口里启动每个`mongod`。

^[2]不要在指定配置服务器地址的时候添加空格。

如果成功，命令应答会包含刚才添加的集群的名字。我们可以检查配置数据库的shards集合来看看是否生效。可以使用getSiblingDB()方法代替use命令来切换数据库：

```
> db.getSiblingDB("config").shards.find()
{ "_id" : "shard-a", "host" : "shard-a/localhost:30000,localhost:30001" }
{ "_id" : "shard-b", "host" : "shard-b/localhost:30100,localhost:30101" }
```

作为一个快捷方式，listshards命令返回相同的信息：

```
> use admin
> db.runCommand({listshards: 1})
```

当我们还在讨论如何返回分片集群的配置信息时，shell的sh.status()命令已经完美地总结了集群的信息。来吧，现在就试试运行它。

12.4.3 分片集合

接下来的步骤是在数据库上启动分片。分片不会自动完成，而是需要在数据库里提前为集合做好设置才行。我们使用的例子数据库叫cloud-docs，所以启动分片的命令如下所示：

```
> sh.enableSharding("cloud-docs")
```

和之前一样，我们可以检查配置数据来看看刚才的修改。配置数据库包含一个名为databases的集合，它包含数据库的列表。每个文档指定了数据库主片的地址以及它是否分区（是否启动分区）：

```
> db.getSiblingDB("config").databases.find()
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "cloud-docs", "partitioned" : true, "primary" : "shard-a" }
```

现在我们要做的就是分片spreadsheets集合。当分片集合时，需要定义一个分片键。这里我们使用组合分片键{username:1, _id: 1}，因为这样便于分布式数据并且易于查看和理解块范围：

```
> sh.shardCollection("cloud-docs.spreadsheets", {username: 1, _id: 1})
```

再说一次，我们可以通过查看分片集合中的配置数据库来检查配置信息：

```
> db.getSiblingDB("config").collections.findOne()
{
  "_id" : "cloud-docs.spreadsheets",
  "lastmod" : ISODate("1970-01-16T00:50:07.268Z"),
  "dropped" : false,
  "key" : {
    "username" : 1,
    "_id" : 1
  },
  "unique" : false
}
```

分片集合的全名

分片集合的分片键

不要过度担心这个文档里的字段。这是MongoDB用来追踪集合的内部元数据，而且用户不会直接访问它。

分片一个空集合

分片集合的定义可能会勾起你一些回忆：看起来有点像索引定义，特别是使用了唯一的键。当分片一个空集合时，MongoDB会在每个分片上创建一个与分片键对应的索引^[1]。要验证这个问题，直接使用shell连接分片并运行getIndexes()命令即可。这里连接第一个分片，然后不出意外，输出结果包含分片键索引的信息：

```
$ mongo localhost:30000
> use cloud-docs
> db.spreadsheets.getIndexes()
[
  {
    "name" : "_id_",
    "ns" : "cloud-docs.spreadsheets",
    "key" : {
      "_id" : 1
    },
    "v" : 0
  },
  {
    "ns" : "cloud-docs.spreadsheets",
    "key" : {
      "username" : 1,
      "_id" : 1
    },
    "name" : "username_1_id_1",
    "v" : 0
  }
]
```

为集合自动创建的_id索引

username 和_id上的组合索引
我们会在这个键上分片

一旦分片了集群，分片就可以工作了。现在向集群中写入数据，数据就会分布式存储到各个分片节点上。下一节我们来看看它是如何工作的。

12.4.4 写入数据到分片集群

我们将会分片集群里插入一些文档，这样就可以观察信息并且移动数据库块，这是MongoDB分片集群的本质。相同的文档，每个表示一个单独的电子表格spreadsheet，如下所示：

```
{
  _id: ObjectId("4d6f29c0e4ef0123afdacaeb"),
  filename: "sheet-1",
  updated_at: new Date(),
  username: "banks",
  data: "RAW DATA"
```

^[1]如果对已经存在的集合分片，就必须在运行 shardcollection 命令之前创建一个与分片键对应的索引。

```
}
```

注意: data 字段会包含一个5 KB的字符串, 用来模拟用户数据。

本书附带的实例代码里包含了本章的写入数据到集群的Ruby脚本。脚本接受一个迭代次数作为参数, 每次迭代都会为200个用户中的每个用户插入一个5 KB文档。脚本代码如下所示:

```
require 'rubygems'
require 'mongo'
require 'names'

@con = Mongo::MongoClient.new("localhost", 40000)
@col = @con['cloud-docs']['spreadsheets']
@data = "abcde" * 1000

def write_user_docs(iterations=0, name_count=200)
  iterations.times do |iteration|
    name_count.times do |name_number|
      doc = { :filename => "sheet-#{iteration}",
              :updated_at => Time.now.utc,
              :username => Names::LIST[name_number],
              :data => @data
            }
      @col.insert(doc)
    end
  end
end

if ARGV.empty? || !(ARGV[0] =~ /\d+$/)
  puts "Usage: load.rb [iterations] [name_count]"
else
  iterations = ARGV[0].to_i
  if ARGV[1] && ARGV[1] =~ /\d+$/
    name_count = ARGV[1].to_i
  else
    name_count = 200
  end
  write_user_docs(iterations, name_count)
end
```

使用 Ruby 驱动
连接 MongoDB

实际插入数据到 MongoDB
数据库的函数

如果拿到代码, 可以直接在命令行里运行, 无参数插入200个值:

```
$ ruby load.rb 1
```

现在通过shell连接mongos。如果查询spreadsheets集合, 就会发现它确实包含200个文档, 而且总大小约1MB。我们也可以查询单个文档, 但要确保排除了例子数据字段(因为我们不想在窗口上打印5KB的文本):

```
$ mongo localhost:40000
> use cloud-docs
> db.spreadsheets.count()
200
> db.spreadsheets.stats().size
1019496
> db.spreadsheets.findOne({}, {data: 0})
{
  "_id" : ObjectId("4d6d6b191d41c8547d0024c2"),
  "username" : "Cerny",
```



```

"updated_at" : ISODate("2011-03-01T21:54:33.813Z"),
"filename" : "sheet-0"
}

```

检查分片

现在我们可以检查智能分片机制。切换到配置数据库，然后检查数据块的数量：

```

> use config
> db.chunks.count()
1

```

目前只有一个数据库。我们来看看它是什么样子：

```

> db.chunks.findOne()
{
  "_id" : "cloud-docs.spreadsheets-username_MinKey_id_MinKey",
  "lastmod" : {
    "t" : 1000,
    "i" : 0
  },
  "ns" : "cloud-docs.spreadsheets",
  "min" : {
    "username" : { $minKey : 1 },
    "_id" : { $minKey : 1 }
  },
  "max" : {
    "username" : { $maxKey : 1 },
    "_id" : { $maxKey : 1 }
  },
  "shard" : "shard-a"
}

```

← 最小字段

← 最大字段

你能猜出来这个块表示的范围吗？如果只有一个数据块，它就会覆盖整个分片集合。通过min字段和max字段来证实，数据块的范围由\$minKey和\$maxKey界定。

minKey 和 maxKey

\$minKey和\$maxKey用在比较操作中，作为BSON类型的边界。BSON是MongoDB数据库的原生数据格式。\$minKey是指小于所有的BSON类型，而\$maxKey是指要大于所有的BSON类型。因为给定字段的值都可以包含任意BSON类型，所以MongoDB使用这两个类型来标记分片集合的数据库数据块起始点。

通过添加更多的数据到spreadsheets集合中，可以看到更有意思的数据分片范围。可以再次使用Ruby脚本，但是这次会运行100次迭代，插入总大小为100 MB的20 000个文档：

```
$ ruby load.rb 100
```

验证插入工作：

```
> db.spreadsheets.count()
```

```
20200
> db.spreadsheets.stats().size
103171828
```

例子插入速度

注意，插入数据到分片集群中要花费几分钟。这主要有2个原因：

- (1) 每个插入都需要客户端执行一个往返请求，生产环境里可以执行海量数据插入。
- (2) 我们只在一台机器上运行所有的分片节点服务器。

这会对磁盘造成巨大的负担，因为4个节点同时在写入数据（两个可复制集的主节点和从节点）。只要在满足要求的生产环境中，插入速度将会更快。

插入这么多数据以后，就会发现多了几个数据块。我们可以通过检查集合中的数据块的数量来验证这个猜想：

```
> use config
> db.chunks.count()
10
```

也可以使用sh.status()命令查看更详细的信息。这个方法会打印所有的数据块信息，并且包含范围信息。为了简单，我们只显示前2个块的信息：

```
> sh.status()
sharding version: { "_id" : 1, "version" : 3 }
shards:
{ "_id": "shard-a", "host": "shard-a/localhost:30000,localhost:30001" }
{ "_id": "shard-b", "host": "shard-b/localhost:30100,localhost:30101" }
databases:
{ "_id": "admin", "partitioned": false, "primary": "config" }
{ "_id": "test", "partitioned": false, "primary": "shard-a" }
{ "_id": "cloud-docs", "partitioned": true, "primary": "shard-b" }
shard-a 5
shard-b 5
{ "username": { $minKey : 1 }, "_id" : { $minKey : 1 } } --> {
  "username": "Abdul",
  "_id": ObjectId("4e89ffe7238d3be9f0000012") }
on: shard-a { "t" : 2000, "i" : 0 }
{ "username": "Abdul",
  "_id": ObjectId("4e89ffe7238d3be9f0000012") } --> {
  "username": "Buettner",
  "_id": ObjectId("4e8a00a0238d3be9f0002e98") }
on : shard-a { "t" : 3000, "i" : 0 }
```

从最小键值
开始的第
一个数据块

从第一个数据块
末尾开始的第
二个数据块

查看多个分片服务器的数据

现在结构已经变了。正如我们在图12.4里看到的，现在有10个数据块。显然，每个数据块表示一个连续范围的数据。

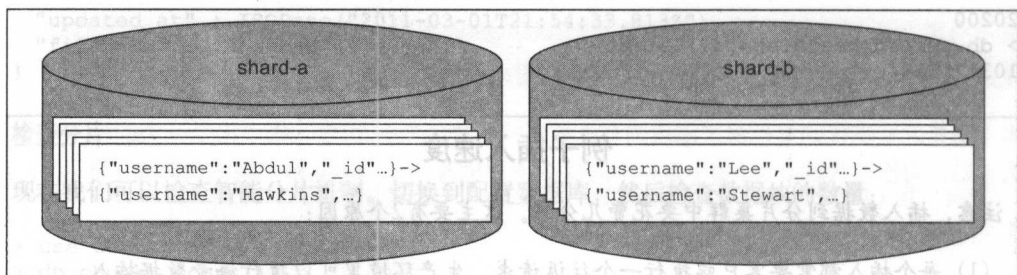


图 12.4 spreadsheets 集合的数据库分布

我们可以在图12.4里看到，shard-a的数据块范围是从Abdul的文档到Buettner的文档，正如我们在输出结果里看到的信息一样。这意味着所有的分片键值介于两个值之间的文档都会插入或者从这个shard-a^[1]片节点的数据块中找到。我们也可以在图中看到，shard-b也有一些数据块，从Lee的文档到Stewart的文档，这意味着任意分片键值位于这之间的文档都会被保存到shard-b上，当然也可以从shard-b上找到。我们也可以使用`sh.status()`输出所有的数据块信息，但是还有更直接的方法：在数据块集合上运行查询，过滤分片的名字，并且计算可以返回多少个文档：

```
> db.chunks.count({"shard": "shard-a"})
5
> db.chunks.count({"shard": "shard-b"})
5
```

只要集群的数据规模小，分割算法决定了分割经常发生。这也是我们现在看到的。这样我们就可以在早期对数据分布做很好的优化。从现在开始，只要其余的写入均匀地分布到各个数据块的范围内，就很少发生迁移。

分割与迁移

表象背后，MongoDB底层依赖2个机制来保持集群的平衡：分割与迁移。

分割是把一个大的数据块分割为2个更小的数据块的过程。它只会数据块大小超过最大限制的时候才会发生，目前的默认设置是64MB。分割是必须的，因为数据块太大就难以在整个集合中分布。

迁移就是在分片之间移动数据块的过程。当某些分片服务器包含的数据块数据量大大超过其他分片服务器时就会触发迁移过程，这个触发器叫做迁移回合(migration round)。在一个迁移回合中，数据块从某些分片服务器迁移到其他分片服务器，直到集群看起来相对平衡为止。我们可以想象一下这两个操作，迁移比分割昂贵得多。

^[1]如果集全参照做了所有这些例子，注意，数据块的分布可能不太一样。

实际上，这些操作不应该影响我们，但是明白这一点非常有用，当遇到性能问题的时候就要想到可能它们正在迁移数据。如果插入的数据分布均匀，各个分片上的数据集应该差不多以相同的速度增长，则迁移应该不会频繁发生。

现在分割的门槛将会增加。通过大量插入数据，我们可以看到分割会降低，以及数据块如何增长到最大值。尝试向分片集群插入800MB数据。再使用一次Ruby脚本，记住，每次迭代插入约1MB的数据：

```
$ ruby load.rb 800
```

这个过程会花费更长的时间。启动插入脚本后，大家可以出去吃个烧烤喝个啤酒再回来继续。在完成以后，数据总量的规模增加到了原来的8倍。但如果检查数据块的状态，就会看到数据块数量仅仅翻倍了，有点意外：

```
> use config
> db.chunks.count()
21
```

数据块越多，平均的数据块范围越小，但是每个数据块会包含越多的数据。例如，集合里的第一个数据块从Abbott到Bender，但是大小已经60 MB。因为默认的最大数据块是64 MB，如果继续插入更多数据，很快就会看到这个数据块发生分割。

另外一个要注意的就是数据分布十分均衡，和之前一样：

```
> db.chunks.count({"shard": "shard-a"})
11
> db.chunks.count({"shard": "shard-b"})
10
```

虽然在最后插入800 MB数据的过程中数据块数量增加了，但是我们可以确信没有发生数据迁移；一种可能的情况是，最初的数据块被分割为2个。可以通过查询配置数据库的changelog来验证此问题：

```
> db.changelog.count({"what": "split"})
20
> db.changelog.find({"what": "moveChunk.commit"}).count()
6
```

这与猜测一致。发生了20次分割，增加了20个数据块，但是只有6次数据迁移。为了更深一步看看究竟发生了什么，我们可以扫描修改日志的项目。例如，这是第一个数据块移动的日志项目：

```
> db.changelog.findOne({"what": "moveChunk.commit"})
{ "_id" : "localhost-2011-09-01T20:40:59-2",
  "server" : "localhost",
  "clientAddr" : "127.0.0.1:55749",
  "time" : ISODate("2011-03-01T20:40:59.035Z"),
  "what" : "moveChunk.commit",
```



这里我们看到数据块从shard-a移动到shard-b。通常，在修改日志里找到的文档都是可读的。当我们学习了更多的分片集群知识时，就可以开始构建自己的分片集群了，配置修改日志提供了分割和迁移行为的最佳参考信息，可以经常查阅。

12.5 分片集群查询和建立索引

Querying and indexing a shard cluster

从应用程序的角度来说，从分片集群查询数据和从单个mongod查询数据是没有区别的。两种情况下查询接口和结果集上的迭代过程是一样的。但是在表面背后，底层操作是不同的，值得我们去深入研究。

12.5.1 查询路由

假设我们在查询分片集群。mongos需要联系多少分片服务器才能返回正确的结果呢？如果想知道一下，就会看到这取决于传递给find查找方法的查询选择器中的分片键表示的范围。记住配置服务维护了分片键值到分片服务器的信息。这些映射会关联到我们之前介绍的数据块。如果查询包含分片键，mongos就可以快速咨询数据块数据来确定它属于哪个分片。这叫做目标性查询。

如果查询的分片键不属于查询，查询集合就必须访问所有的分片服务器以填充查询。这个过程叫做全局或分散/集中查询。

图12.5解释了两种不同的类型。

图12.5所示中包含2个分片集群、2个mongos路由，以及2个应用服务器。集群的分片键是{username: 1, _id: 1}。我们将会在12.6节里讨论如何选择一个好的分片键。

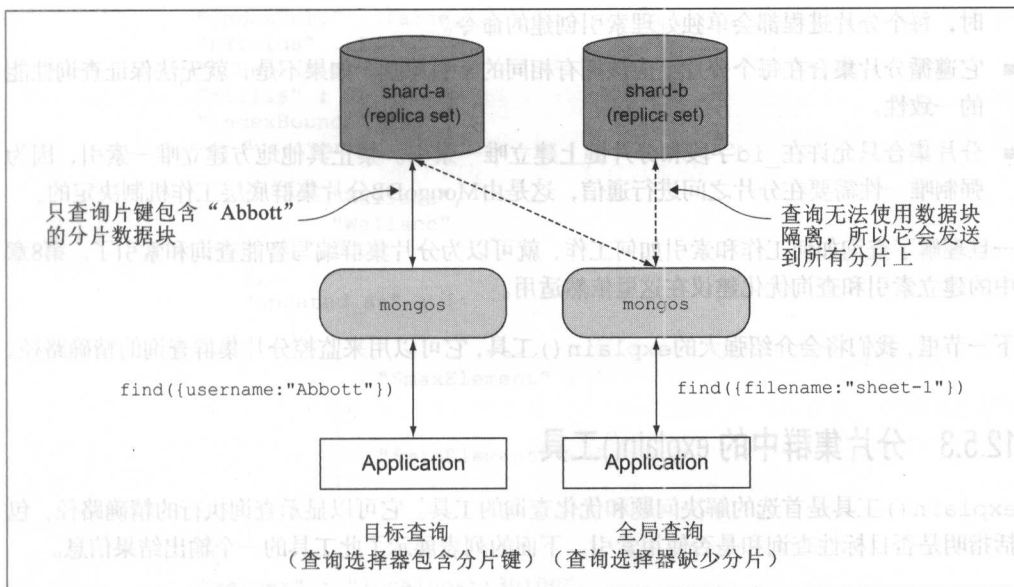


图 12.5 分片集群里的目标和全局查询

在图示的左边，可以看到查询选择器包含username的目标查询。此时，mongos路由可以使用username字段直接把查询转发到正确的分片节点。

在图示的右侧，可以看到全局或分散/集中查询，在查询选择器中它不包含分片键的任意部分。此时，mongos必须广播查询到两个分片中。

目标查询的性能不能夸大。如果所有的查询都是全局的，这意味着每个分片节点都必须响应集群的每个查询。相反如果所有的查询是目标性的，每个分片只需要处理与自己相关的请求。伸缩性的含义非常清晰。

但是定位并非是影响分片集群性能的唯一因素。正如我们会在下一节里看到的，每个在未分片部署下性能问题都会适用于分片集群，只是分散到单独的机器上。

12.5.2 分片集群中建立索引

无论插入定位多么完美，最终都必须运行在一个分片上。这意味着，如果分片响应查询慢，则集群也会慢。

未分片模式下，索引是优化查询性能的重要手段。

关于分片集群的索引只需要记住以下几个要点：

- 每个分片维护自己的索引。当在分片集合上声明索引时，每个分片都会为自己的集合部分定义单独的索引。例如，当连接mongos使用db.spreadsheets.createIndex()命令

时，每个分片进程都会单独处理索引创建的命令。

- 它遵循分片集合在每个分片上应该拥有相同的索引原则。如果不是，就无法保证查询性能的一致性。
- 分片集合只允许在 `_id` 字段和分片键上建立唯一索引。禁止其他地方建立唯一索引，因为强制唯一性需要在分片之间进行通信，这是由 MongoDB 分片集群底层工作机制决定的。

一旦理解了查询如何工作和索引如何工作，就可以为分片集群编写智能查询和索引了。第8章中的建立索引和查询优化建议在这里依然适用。

下一节里，我们将会介绍强大的 `explain()` 工具，它可以用来监控分片集群查询的精确路径。

12.5.3 分片集群中的 `explain()` 工具

`explain()` 工具是首选的解决问题和优化查询的工具。它可以显示查询执行的精确路径，包括指明是否目标性查询和是否使用索引。下面的列表展示了此工具的一个输出结果信息。

列表12.1 索引并查询以返回一个用户更新的最新文档。

```
mongos> db.spreadsheets.createIndex({username:1, updated_at:-1})
{
  "raw" : {
    "shard-a/localhost:30000,localhost:30001" : {
      "createdCollectionAutomatically" : false,
      "numIndexesBefore" : 3,
      "numIndexesAfter" : 4,
      "ok" : 1
    },
    "shard-b/localhost:30100,localhost:30101" : {
      "createdCollectionAutomatically" : false,
      "numIndexesBefore" : 3,
      "numIndexesAfter" : 4,
      "ok" : 1
    }
  },
  "ok" : 1
}
mongos> db.spreadsheets.find({username: "Wallace"}).sort({updated_at:-1}).explain()
{
  "clusteredType" : "ParallelSort",
  "shards" : {
    "shard-b/localhost:30100,localhost:30101" : [
      {
        "cursor" : "BtreeCursor username_1_updated_at_-1",
        "isMultiKey" : false,
        "n" : 100,
        "nscannedObjects" : 100,
        "nscanned" : 100,
        "nscannedObjectsAllPlans" : 200,
        "nscannedAllPlans" : 200,
        "scanAndOrder" : false,
```

updated_at 和 username
中用来获取文档的索引


```
    "millis" : 4  
  }
```

我们从`explain()`计划里看到,这个查询只发送给了一个分片节点❶,而且在那个节点上使用创建的索引进行高效查询❷。注意,`explain()`计划输出结果是从v2.6开始支持,并且在v3.0以后的版本里做了修改。第8章包含了在MongoDB 3.0版本里使用`explain()`命令的输出结果。如果有任何疑问,可以查看官方文档<https://docs.mongodb.org/manual/reference/method/cursor.explain/>。

12.5.4 分片集群中聚合

值得注意的是,也可以从分片集群中获益。聚合的分析比单个查询复杂一些,而且当使用优化时可能版本支持的还不同。幸运的是,聚合框架也有个可以监控查询执行详细信息的`explain()`工具。作为基本原则,聚合查询需要联系的分片服务器数量依赖于操作需要数据的分布情况。例如,如果计算整个数据库中的文档数量,就需要查询所有的分片节点。但是如果只计算一个小范围内的文档数量,可能就不需要查询每个分片节点。具体的差别可以参考最新的官方文档<https://docs.mongodb.org/manual/reference/method/db.collection.aggregate/>。

12.6 选择分片键

Choosing a shard key

在12.3节里,我们看到了如何使用分片键把集合分割为多个逻辑范围,也就是数据块。在12.5.1节里,我们看了mongos如何使用这些信息来定位文档所在的位置。

本节里,我们将会深入讨论至关重要的分片键选择过程。糟糕的分片键会阻止应用程序使用分片提供的许多好处。严重情况下,查询和插入的性能都会受到严重的影响。这也增加了决策的严重性,一旦选择了分片键,就只能坚持使用它。分片集群的键是不变的^[1]。

理解糟糕的分片键设计带来的坏处的最好方式就是一步一步找个好的分片键,并深入分析每个分片键值。这就是接下来我们要做的,继续使用之前的spreadsheet应用程序作为例子。

在我们为spreadsheet应用程序找到最佳的分片键以后,在本章的结尾部分,我们将会思考如何为邮件系统选择分片键,这里有什么区别。这里会强调最佳分片键的选择依赖于具体的应用程序。

在我们为spreadsheet应用选择最佳分片键的过程中,会看到3个主要的陷阱:

^[1] 注意一旦创建分片键以后就没有修改它的好办法。最好的办法是使用新的分片键创建一个新的分片集合,然后从旧的集合里导出到新的分片集合中。

- **热点** 某些分片键会导致所有的读或者写都操作在单个数据块或单个分片上。这可能导致单个分片服务器严重不堪重负，而其他分片服务器闲置，无所事事。
- **不可分割数据块** 过于粗粒度的分片键可能导致许多文档使用相同的分片键。因为分片是基于分片键值的范围，所以意味着这些文档不能被分割为多个数据块，这个最终会限制MongoDB均匀分布数据的能力。
- **糟糕的定位** 即使写压力可以完美分布到集群中，如果我们的分片键与某些查询没有关联，也会导致糟糕的查询性能。我们在12.5.1小节已经看过全局和目标查询这些内容了。

现在来看看如何为spreadsheet应用找到最佳的分片键，这些实际情况是如何出现的。

12.6.1 非平衡写入(热点)

第一个可能想到的分片键是{ "_id" : 1 }，它会在_id字段上进行分片。

乍一看，_id字段好像是最佳的候选：它肯定会出现每个文档中，默认就有索引，在许多查询里会使用到，而且MongoDB会使用BSON Object ID类型自动生成它，本质上是GUID(全局的唯一标识符)。

但是使用Object ID作为分片键值有个显著的问题：它的值是严格升序的。这意味着每个新的文档必须有一个比集合中其他文档大的分片键值。为什么这会是个问题呢？因为，如果系统已经完全平衡了，意味着每个新的文档都会进入新的数据块中，这个值很快就会达到\$maxKey。这个问题最好通过例子理解，如图12.6所示。

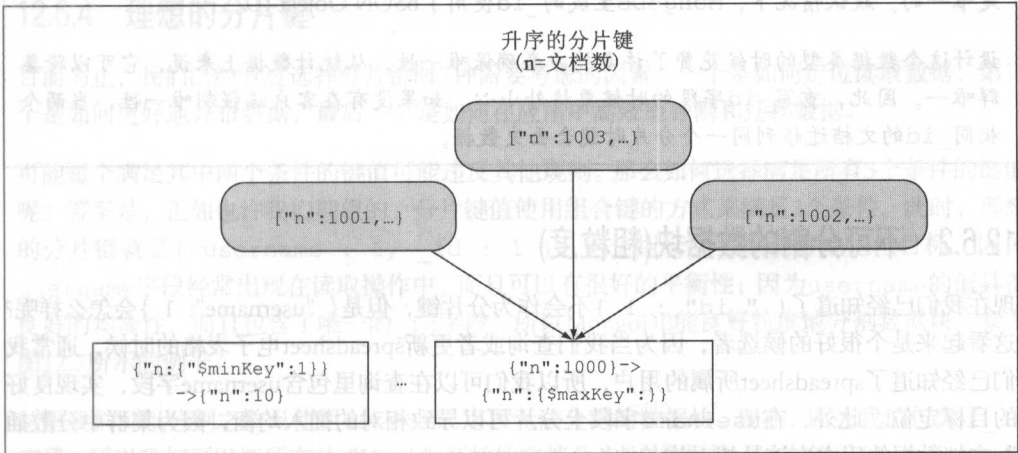


图 12.6 升序的分片键值导致所有的写入都进入单个数据块

为了简单起见，这个例子使用了存储在字段n里的文档数字作为分片键而不是BSON Object ID。

正如所见，我们已经插入了1到1000的文档，现在将要插入1001、1002、1003三个文档。因为MongoDB还没有见到大于1000的号码，所以没有理由来分割从1000到\$maxKey的数据块。这意味着新的3个文档也全部属于当前的数据块，而且所有的新文档都是这样的。

要看看它如何影响真实的应用，就考虑使用剩余BSON Object id作为分片键。实际上，这意味着每个新创建的spreadsheet文档都属于同一个数据块，也就是每个新的文档都被写入到单个片中。这也是性能问题的根源所在。即使有几百个分片，所有的写入都会发送到单个片上而其他片空闲着。最重要的是，MongoDB会全力以赴从过载的片上迁移数据，这样就更加剧了性能恶化。

这也大大抵消了分片的最大优势：跨机器自动分布插入负载压力^[1]。所以说，如果仍然想继续使用_id字段作为分片键，则有2个选择：

- 使用自定义非自增的标识符重写_id字段。如果这样做，记住，即使在分片集群里，_id也必须唯一。
- 在哈希键值上设置分片键。这样会告诉MongoDB使用哈希函数的结果作为分片键，而不是直接使用分片键。哈希函数用来产生随机结果，它可以确保插入更加均匀地分布在集群中。但是这也意味着范围查询需要扫描多个分片，因为虽然分片键的值相似，但是它们的哈希值完全不同。

唯一性陷阱

MongoDB只能确保分片键值是唯一的。它不能在任意键上强制唯一性，因为一个分片无法通过检查另外一个分片来看看是否有重复值。令人惊讶的是，这也包括_id字段，它必须是唯一的。默认情况下，MongoDB生成的_id使用了BSON Object ID。

设计这个数据类型的时候花费了许多精力来确保唯一性，从统计数据上来说，它可以跨集群唯一。因此，重写_id字段的时候要格外小心。如果没有在客户端强制唯一性，当两个相同_id的文档迁移到同一个分片时就会丢失数据。

12.6.2 不可分割的数据块(粗粒度)

现在我们已经知道了{ "_id" : 1 }不会作为分片键，但是{ "username": 1 }会怎么样呢？这看起来是个很好的候选者，因为当我们查询或者更新spreadsheet电子表格的时候，通常我们已经知道了spreadsheet所属的用户，所以我们可以查询里包含username字段，实现良好的目标定位。此外，在username字段上分片可以导致相对的插入均衡，因为集群中分散插入文档数据的压力应该是相对均匀的。

^[1]注意，只要文档是随机更新的，升序的分片键不应该影响更新。

使用这个字段只有一个问题：它如此粗粒度，可能导致的极端情况就是数据块无限增长。要弄明白这个问题，假想一下用户“Verch”要插入10 GB的spreadsheet电子表格数据。这会让包含username为“Verch”的文档包含超过64MB最大值的数据块。

正常情况下，当数据块变得太大时，MongoDB可以把它们分割为较小的数据块，然后跨集群进行平衡迁移数据。但是，此时已经没有空间可以分割数据块了，因为它只包含了一个单独的分片键值。这会导致许多技术问题，但是最终的结果就是集群变得不平衡了，而且MongoDB无法再进行高效的平衡操作。

12.6.3 糟糕的定位(分片键不在查询中)

在看了这些问题之后，你可能会想，“好吧，那我就只能选择完全随机的分片键值。它是唯一的并且不是升序的”。虽然它可以解决写入的均衡问题，但是如果你打算从集群中读取数据，则它不是一个好的分片键。正如我们在12.5.1一节看到的，如果它们包含分片键值，则查询只能被路由到正确的分片上。如果分片键完全是随机的，在查询文档的时候，我们就不知道分片键值是什么意思。可能大家也理解了为什么ID和username这种字段会成为分片键值的首选了。这意味着路由必须全局转发，这样可能会影响到性能。

这就是说，如果编写的应用要做的工作只是保存大量的数据而不需要查询，例如，收集大量传感器数据以后再批处理分析的应用系统(处理数据时大部分都需要查询所有的分片)，则随机的唯一的分片键值或许不是个坏主意。事实上，如果可以确保唯一性(通过生成GUID)，并且确保不是升序，则可以重写_id字段。

12.6.4 理想的分片键

目前为止，我们已经看过选择分片键时3种需要考虑的因素。一个是如何定位读取数据，第二个是如何更好地分布数据，最后一个是如何在应用中高效地分割和迁移数据。

可能每个满足其中两个条件的键值可能违反其他规则。那么如何选择满足所有3个条件的键值呢？答案是，正如也许我们期望的，分片键值使用组合键的方式来满足3个条件。此时，理想的分片键就是{ username : 1, _id : 1 }。这个组合键可以很好地定位目标，因为username字段经常出现在读取操作中，而且可以有很好的平衡性；因为username的值具备良好的均衡性，而且包含了唯一的_id字段，所以MongoDB能良好粒度地分割数据块。参考图12.7所示的例子。

这里我们可以分割之前只使用username字段无法分割的数据块。现在，因为我们包含了_id字段，所以我们可以把所有从“Verch”开始的文档分为2个数据块，这在使用{ "username": 1 }作为分片键时是无法完成的。

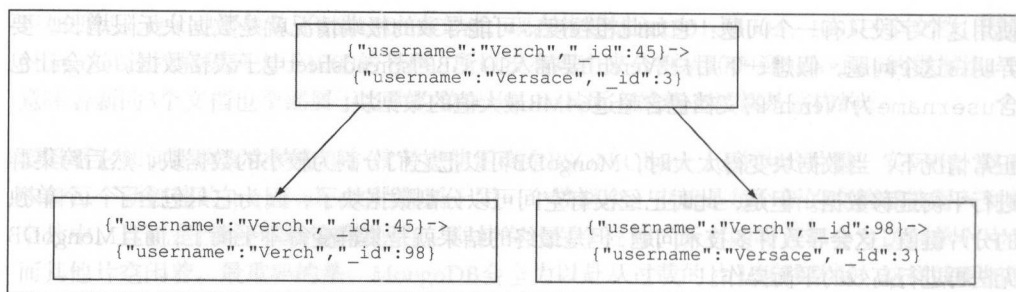


图 12.7 分割只使用 username 字段无法分割的数据块

索引问题

本章讨论如何选择高效的分片键，但是要记住，索引和单节点服务器一样重要。正如我们在12.5.2小节讨论的，即使分片键非常完美，而且每次查询都会路由到单个片上，如果性能在每个片上都很差，则整体的性能也不会好。因此，在集合分片之前MongoDB需要在分片键上创建索引。这里没有技术原因，除了不自动完成就会出现常见错误的问题。当设计分片系统时，请记住之前在单节点服务器上学习的所有知识。这在分片集群中同等重要，而且可能更重要，因为我们在更大的集群中操作。

一个重要的考虑因素是索引位置。它指的是索引中后续插入的临近度。此时，随机插入执行的就比较糟糕，这是因为整个索引都会被加载进入内存；而顺序插入执行的就比较好，因为只需要索引的末尾在内存里。这是非直接对比分片键不是升序的时候的需求，说明当设计分片集群系统时恰当的索引和最佳分片键值选择都需要单独关注。

幸运的是，我们之前选择的分片键{ "username" : 1, "_id" : 1 }满足了所有的需求。它可以在username字段上实现良好的分布式存储，所以从集群的角度来说是均衡的；而且它在_id字段上是升序的，所以它对于每个username又是升序的，因此提供了优秀的索引位置。

12.6.5 设计折中(email 应用)

在分片中，有时候还有一些必须处理的固有的设计折中问题。

我们的spreadsheet例子是干净的分片集群例子，因为两个读/写都是通过username关联的，所以之前选择的分片键在两者之间都提供了良好的性能。但是如果我们需要查询的字段与写入模式不相关呢？例如，如果我们要建立一个email程序，提供发送和读取邮件的邮箱。乍一看这不是问题，直到我们意识到写入和发送者相关、读取者和接收者相关。因为从理论上讲，任何人都可以给任何人发邮件，而且很难预测两个字段之间的关系。在这一节里，我们将会思考两个方法。首先，我们将会使用更加简单的方法，基于发送者分片；然后来看看如何基

于接收者分片，改变一下使用模式。这两个方法如图12.8所示。

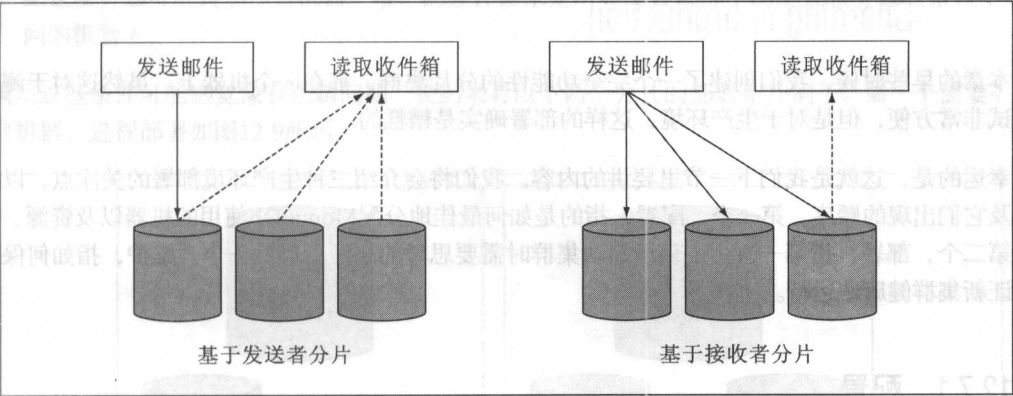


图 12.8 应用可能的两种分片方式概要图

基于发送者分片

第一个方法基于发送者分片。这看起来是个敏感的选择，因为每个邮件都只有一个发送者。此时，我们的写模式非常好：每个写入都只会写入一个分片。但是当需要读取用户收件箱的时候会发生什么呢？如果我们基于发送者分片，则接收者的所有邮件都会碎片化存储，我们需要进行全局查询，如图12.8所示，因为我们不知道某个用户的邮件来源。

基于接收者分片

下一个策略就是基于接收者分片。这个有点复杂，因为每个邮件必须写入到多个地址以确保每个接收者都收到了邮件的拷贝，如图12.8所示。这非常不幸，意味着我们必须花费更长的时间，并且向集群添加更多的负载。但是这个方法有个好处：读者收件箱的查询定位很好，可以很快返回并且容易伸缩。

图12.8右边所示的方法是什么？很难精确回答，但是对于这个应用来说，第二个方法更好一些。有两个原因。一个是用户也许读收件箱比发送者更频繁，另外一个便于用户概念化需要实际工作的邮件，比如发送邮件，但是还没有来得及读取收件箱的工作。你多久听到一次“我们无法打开收件箱！”这件邮件应用最容易做到的工作？最后，这些都是推测，与真实的应用一样，需要仔细衡量和测试以确定实际的使用模式。

这个例子要说明的就是，分片键依赖于专门的应用。没有万能的分片键值可以覆盖所有的用例，所以记住，并思考自己期望的读写模式是什么。

到目前为止，我们已经详细讨论了分片集群背后的理论知识，还有如何配置这些参数来优化系统的性能。下一节里，我们将会详细介绍配置真实的分片集群时要考虑的问题。

12.7 生产环境下分片集群

Sharding in production

本章的早些时候，我们创建了一个完全功能性的分片集群，都在一个机器上。虽然这对于测试非常方便，但是对于生产环境，这样的部署确实是糟糕的。

幸运的是，这就是我们下一节里要讲的内容。我们将会介绍三种生产环境部署的关注点，以及它们出现的顺序。第一个，配置，指的是如何最佳地分配MongoDB使用的机器以及资源。第二个，部署，指第一次生产环境启动集群时需要思考的东西。最后一个，维护，指如何保证新集群健康地运行。

12.7.1 配置

部署系统时候第一个要考虑的事情就是配置，或者说是如何为MongoDB分配资源和机器。

部署拓扑

要启动MongoDB例子分片集群，就必须启动总共9个进程（每个分片3个mongods，加上3个配置服务器）。数字有点吓人！假设在生产环境里运行两个分片集群，这样需要9个独立的服务器。幸运的是，实际需要的数量更少。我们可以通过查看集群每个组件的期望资源需求来了解原因。

第一个考虑可复制集。每个可复制集成员包含分片数据的完全拷贝，要么是主节点要么是从节点。这些进程通常需要足够的磁盘空间来存储数据副本，需要足够的内存来高效处理数据。因此，此时分片集群中的mongod进程对资源是最敏感的，必须有独立的机器。

可复制集中的裁判呢？这些进程与可复制集中的其他成员一样，除了不复制数据只保存配置信息，这些都是最小的数据量。因此裁判进程压力较小，可以不需要独立的服务器。

接下来是配置服务器。它们存储的也是相对较小的数据。例如，配置服务器管理的例子中可复制集数据总共约30KB。假设分片集群的数据线性增长，那么1 TB的分片集群大概配置30 MB数据^[1]。这意味着配置服务器不需要自己独立的机器。一些用户把配置服务器的关键作用放到低配置的服务器上（或者虚拟机上）了。

根据我们已知的可复制集和分片集群知识，可以列出部署需要的最小配置清单：

- 每个可复制集成员，无论是主从节点还是裁判，都需要部署到不同的机器。
- 每个可复制集成员需要自己的机器。
- 可复制集中裁判足够轻量级，可以与其他进程共享机器。回到第10章参考更多关于裁判的

^[1]这是非常保守的估算。真实的值可能更小。

内容。

- 配置服务器可选共享一台机器。唯一的硬件需求是所有配置集群中的配置服务器驻留在不同的机器上。

满足这些条件可能感觉像有逻辑问题。我们来看以下两个分片的部署拓扑例子。第一个需要4台机器，进程部署如图12.9所示。

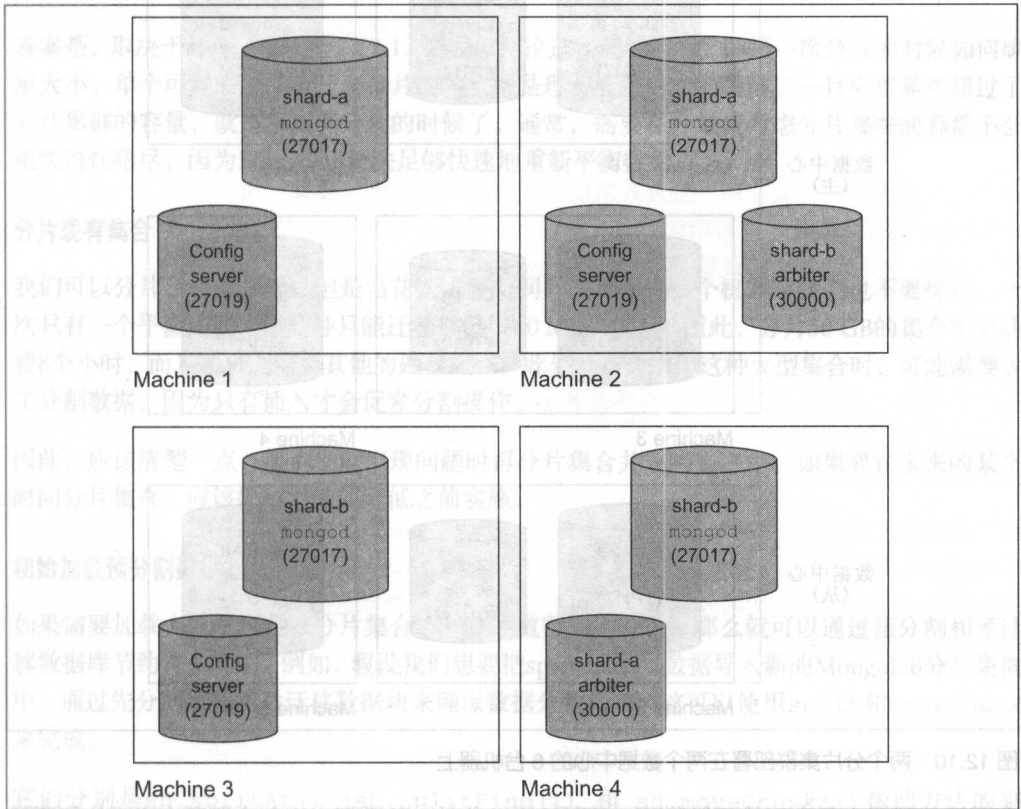


图 12.9 4 台机器两个分片集群的拓扑结构

这个配置满足上面的所有部署规则，每台机器上最主要的部分是分片进程，所有成员分布在不同的机器上。谈到容错问题，这个拓扑图可以容忍任意一个机器出错。无论哪个机器出错，机器都可以正常读/写。如果故障机器是某一个配置服务器，则所有的数据库都会被分割并迁移^[1]。后续的分片操作不会阻止机器继续服务其他操作；分割和迁移可以等到丢失的机器恢复以后进行。

这是两个分片集群的推荐最小配置。对于应用要求最高的可用性和最快的恢复速度，就需要更强的配置。正如前一章讨论的，可复制集由2个复制节点和1个裁判组成。3个节点减少了集

^[1] 所有3个配置服务器需要在任意操作的时候保持在线。

群的脆弱性，而且保证1个节点作为从节点实施。数据中心灾备。图12.10所示为一个更强壮的配置拓扑结构。每个可复制集包含3个节点，每个节点包含完整的数据副本。为了灾难恢复，配置服务器和每个分片中的一个节点都部署在从数据中心。若要想确保这些节点永远不会变为主节点，可以设置它们的优先级为0。

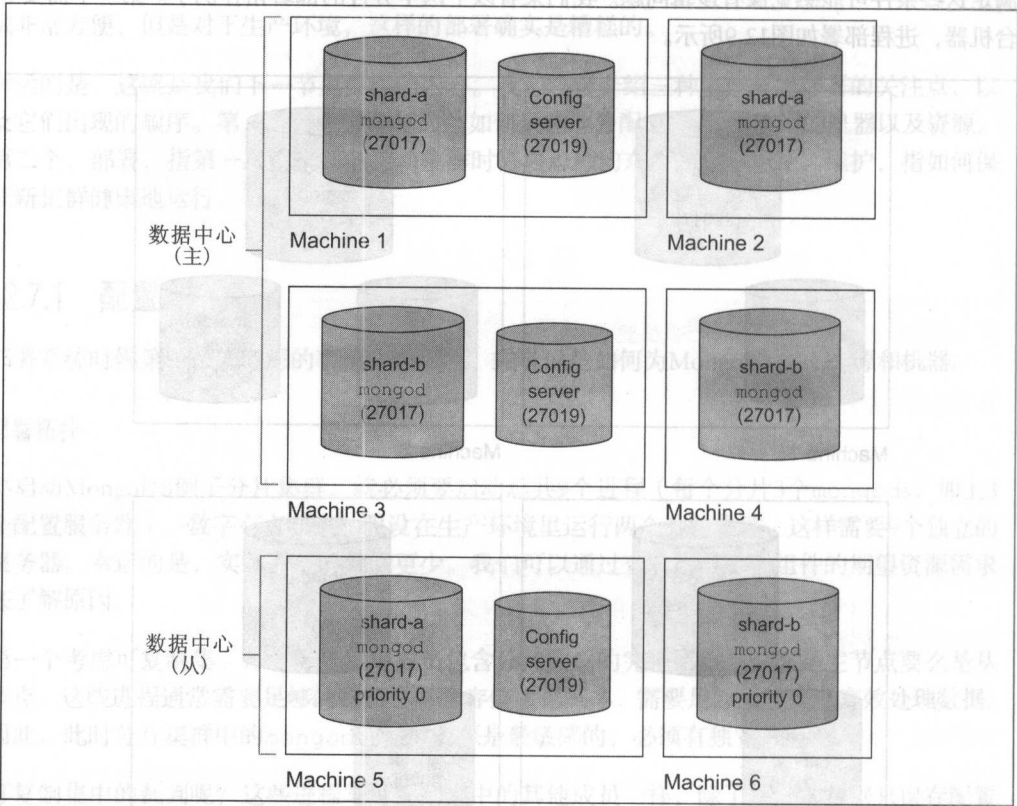


图 12.10 两个分片集群部署在两个数据中心的 6 台机器上

有了这些配置，每个分片都复制2次，不是1次。此外，从数据中心包含了所有灾难恢复所需要的数据，当主数据中心故障时，可以重建分片集群。

哪个分片集群的拓扑图是最好的，这取决于你的应用系统可以容忍多久的宕机时间，即通过恢复时间目标性(RTO)衡量。思考一下潜在的失败场景并且模拟一下，思考一下数据中心灾难的后果。

12.7.2 部署

既然我们已经学习了集群的拓扑架构，现在就来讨论实际的部署和配置了。

生产环境的分片集群配置严格遵守之前12.4一节讲述的例子配置步骤。这里我们介绍关注其

他生产环境部署集群的变量。

添加新分片

用户经常问的问题就是，到底要部署多少分片？每个分片应该多大才好？理所当然地，每个额外的分片都会引入更多的复杂性，而且每个分片都需要复制节点。因此，最好是部署最小数量的大分片而不是较多数量的小分片。但是问题依然存在，每个分片实际应该多大？

答案是，取决于环境。事实上，12.1.2节我们讨论过相同的概念，即第一次分片的时候如何确定大小。单个可复制集或者一个分片的容量就是理解应用需求的关键。一旦应用需求超过了分片集群的容量，就是添加新分片的时候了。通常，需要在开始就考虑分片集群的容量不会很快消耗殆尽，因为MongoDB无法足够快速地重新平衡数据。

分片现有集合

我们可以分片已有的集合，但是当花费很多时间分散数据到多个机器时我们也不要惊讶。一次只有一个平衡回合，每分钟只能迁移100~200 MB的数据，因此，分片50 GB的集合将会花费8个小时，而且还可能需要其他的磁盘操作。此外，当初始化这种大型集合时，可能需要人工分割数据，因为只有插入才会促发分割操作。

因此，应该清楚一点，等到性能出现问题时再分片集合并非明智之举。如果要在未来的某个时间分片集合，应该提前在性能降低之前实施。

初始加载预分割数据

如果需要加载大数据集进入分片集合，并且知道数据的分布，那么就可以通过预分割和预迁移数据库节约许多时间。例如，假设我们想要把spreadsheet 数据导入新的MongoDB分片集群中。通过先分割数据然后迁移数据块来确保数据分散均匀，这可以使用split和moveChunk来完成。

它们分别是sh.splitAt() (sh.splitFind()) 和 sh.moveChunks() 帮助方法的别名。

这里是一个手动分割数据块的例子。使用split命令，指定要分割的集合，然后指定分割点：

```
> sh.splitAt( "cloud-docs.spreadsheets",  
{ "username" : "Chen", "_id" : ObjectId("4d6d59db1d41c8536f001453") })
```

运行此命令时，这个命令会定位数据块中的username为Chen，并且_id为ObjectId("4d6d59db1d41c8536f001453")的文档^[1]。

然后开始在这个点上分割数据块。我们可以这样一直分割下去，直到有分散均匀的数据块集合为止。我们肯定想确保所有创建的数据块都在64MB以内。如果想要加载1GB的数据，就应

^[1] 这种文档不需要存在，因为我们开始分割数据块的集合是个空集合。

该创建20个左右的数据块。

第二步就是确保所有的分片大致上有相同的数据块数目。因为所有的数据块最初都在同一个分片中，后面需要移动它们。每个数据块都可以使用moveChunk命令移动。此帮助方法简化了这个工作：

```
> sh.moveChunk("cloud-docs.spreadsheets", {username: "Chen"}, "shardB")
```

这个代码表示我们想把包含{username: "Chen"}的文档移动到分片B服务器中。

12.7.3 维护

本章结束部分我们将会介绍一些分片集群维护与管理的知识。注意，所有这些工作都可以使用MongoDB官方的监控与自动化工具，我们会在第13章里讨论。这里我们会介绍集群底层发生的事情，因为这些非常重要。MongoDB自动使用了底层命令来实现这些功能。

监控

分片集群是个复杂的组织，我们应该紧密监控它。serverStatus和currentOp()命令可以运行在任意的mongos上，它们的输出结果反映的是分片集群的统计信息。我们将会在下一章里详细讨论这些命令。

除了这些聚合服务器统计信息，我们还想观察数据分布和单独数据块的信息。正如我们在例子集群里看到的，所有这些信息都存在配置数据库里。如果探测到不平衡的数据块或者未检查的数据块增长，可以使用split和movechunk命令来处理这些问题。也可以通过查询日志来看看平衡操作是否因某些原因阻塞了。

手动分区

许多情况下，我们想要在运行的分片集群中手动分割和迁移数据。例如，对于MongoDB 2.6来说，均衡器无法直接处理任意一个分片的压力。显然，一个分片写入的数据越多，它的数据块就越大，并且很可能最终要迁移数据。虽然通过迁移数据块减轻某个分片压力的方案并不难，但这也是movechunk命令起作用的另外一个情况。

添加分片

如果确定需要更多的容量，就可以使用相同的方法来为现有的分片集群添加新的分片服务器。

```
sh.addShard("shard-c/rs1.example.net:27017,rs2.example.net:27017")
```

当用这种方式添加新的分片服务器时，要知道迁移数据花费的时间。正如之前说过的，可以期望数据每分钟迁移100~200 MB。这意味着如果需要添加容量到分片集群中，就应该在性能受影响之前尽早操作。要确定什么时候添加新分片，考虑一下数据的增长速度。显然，我们想要保住索引和工作集位于RAM中。最好在分片服务器的RAM使用率达到90%的前几周就

开始添加新分片服务器。

如果不提前进行安全处理,就可能面临痛苦的境地。一旦索引和工作集无法加载到RAM里,应用就会出现卡死,特别是应用需要高读/写并发吞吐量时。数据库需要反复从磁盘分页加载数据,这会降低读/写速度。此时,添加容量比较困难,因为迁移数据块会增加现有分片服务器的压力。显然,当数据库过载时,最后要做的事情就是增加负载。

所以这里要强调的就是应该监控集群,并且在迫切需要之前添加新容量。

删除分片

很少情况下,或许要删除某个分片。可以使用removeshard命令来实现:

```
> use admin
> db.runCommand({removeshard: "shard-1/localhost:30100,localhost:30101"})
{
  "msg" : "draining started successfully",
  "state" : "started",
  "shard" : "shard-1-test-rs",
  "ok" : 1 }
```

这个命令的返回信息表示数据块正在从这个分片服务器迁移到其他分片上。我们可以再运行一次这个命令来检查搬迁的进度:

```
> db.runCommand({removeshard: "shard-1/localhost:30100,localhost:30101"})
{
  "msg" : "draining ongoing",
  "state" : "ongoing",
  "remaining" : {
    "chunks" : 376,
    "dbs" : 3
  },
  "ok" : 1 }
```

一旦分片被清空,就需要再次确保没有数据块的主分片是我们要删除的分片。我们可以通过查询config.databases集合来检查数据库分片的关系:

```
> use config
> db.databases.find()
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "cloud-docs", "partitioned" : true, "primary" : "shardA" }
{ "_id" : "test", "partitioned" : false, "primary" : "shardB" }
```

shardB 拥有cloud-docs数据库而shardA拥有test数据库。因为正在删除shardB,所以需要修改测试数据库的主节点。我们可以使用moveprimary命令来实现:

```
> db.runCommand({moveprimary: "test", to: "shard-0-test-rs" });
```

运行这个命令是为每个主节点删除分片的数据库。然后运行removeshard来验证数据是否删除完毕:

```
> db.runCommand({removeshard: "shard-1/localhost:30100,localhost:30101"})
{ "msg": "remove shard completed successfully",
  "stage": "completed",
  "host": "localhost:30100",
  "ok" : 1 }
```

一旦删除工作完成，就可以安全地下线这个分片服务器了。

未分片集合

虽然可以删除分片服务器，但是还没有官方的方法可以把一个分片的集合重新复原为不分片状态。如果需要对集合取消分片，最好的方式是，将数据重新导入一个具有新名字的新集合中^[1]，然后使用dorp删除原有的分片集合。例如，假设分片的集合为foo，可以通过连接mongos，使用mongodump工具来获取foo的镜像文件：

```
$ mongodump -h localhost --port 40000 -d cloud-docs -c foo
connected to: localhost:40000
DATABASE: cloud-docs to dump/cloud-docs
cloud-docs.foo to dump/cloud-docs/foo.bson
100 objects
```

把集合数据导出到名为foo.bson的集合文件里。然后使用mongorestore恢复数据：

```
$ mongorestore -h localhost --port 40000 -d cloud-docs -c bar
Tue Mar 22 12:06:12 dump/cloud-docs/foo.bson
Tue Mar 22 12:06:12 going into namespace [cloud-docs.bar]
Tue Mar 22 12:06:12 100 objects found
```

一旦把数据迁移到未分配的集合中，就可以安全地删除旧的分片集合foo了。但是，删除集合的时候要格外小心，因为可能发生问题。必须确保删除对的集合。

备份分片集群

正如我们将会在第13章里看到的，对于备份MongoDB还有一些参数。

绝大部分情况下，这些策略也可以用来备份分片集群中的每个分片。无论使用哪个方法来备份集合，都有2个步骤必须在备份分片集合之前完成：

- 备份分片集群数据的时候，要注意的第一件事情就是可能会发生数据迁移。这意味着，除非大家备份的东西都是在相同的时间点上（这几乎不可能），否则会丢失一些数据。我们将会在本节里介绍如何禁用数据块迁移。
- 备份分片集群时，也必须备份配置服务器元数据。因此，可以指定单配置服务器备份，因为所有的配置服务器的数据都是一样的。与备份分片一样，这应该在数据块迁移禁用之后再做，避免数据丢失。

^[1] 用来导出和恢复数据的工具是mongodump和mongorestore，下一章会介绍。

所幸有内置的机制可以禁用自动化数据块迁移。分片之间所有迁移都是通过一个叫均衡器的进程处理的。一旦停止这个进程，就可以确保不会自动迁移数据了。我们也可以手动促发迁移或者创建新的数据库，但是可能会破坏正常的备份工作，所以确保没有其他进程在运行并做管理操作。

停止并启动均衡器

要禁用均衡器，就使用`sh.stopBalancer()`帮助方法：

```
> use config
> sh.stopBalancer()
```

注意：这要花费很长时间来完成。因为帮助方法只会标记均衡器为禁用，而且不会中断现有的均衡工作。这意味着它必须等待进程间的均衡工作结束。要启用均衡器，可以使用`sh.startBalancer()`帮助方法：

```
> use config
> sh.startBalancer()
```

可以从MongoDB官方文档查询到更多关于均衡器的配置信息，包含了在特定时间内启用均衡器的设置过程。

故障转移和恢复

虽然我们已经介绍了通常的可复制集故障，但是掌握分片集群故障恢复的最佳实践也非常重要。

分片成员故障

每个分片都由一个可复制集组成。如果任意一个成员故障，从服务器就会提升为主服务器，并且mongos会自动连接它。第11章介绍了恢复故障可复制集群的步骤。这个方法依赖于成员是如何故障的，但是无论是不是分片集群，可复制集的故障恢复指令都是一样的。

配置服务器故障

分片集群需要3台配置服务器，但是最多只能2台故障。无论什么时候少于3台配置服务器，其余的配置服务器就会变成只读的，而且所有的分割和均衡操作都会挂起。注意，这不会影响集群的完整性。集群都可以正常读/写，而且均衡器会在3个配置服务器恢复以后重新启动。

要恢复配置服务器，就要从现有的配置服务器拷贝数据到故障配置服务器，然后重新启动服务器^[1]。

13.1.1 集群拓扑

^[1]通常，在复制任意数据文件前，请确保已经锁住 mongod (正如第 11 章描述) 或者完全关闭它。不要在服务器存活状态下拷贝数据文件。

路由服务器故障

无需担心Mongos进程的故障。如果在应用服务器上托管运行mongos，并且mongos发生了故障，则最可能是应用服务器发生了故障。此时的恢复非常简单，就是重启服务器。

无论mongos是如何发生故障的，这个过程都没有自己的状态。这意味着恢复mongos就是重启mongos进程并连接配置文件。

12.8 总结

Summary

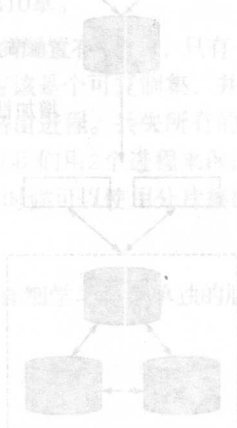
分片集群是在大数据集上维护高性能读/写的有效策略。MongoDB分片集群在许多生产环境里工作良好，当然也可以为我们所用。我们可以利用MongoDB内置的分片机制来代替我们自己半生不熟的分片方案。如果遵循本章中的建议，则对于推荐的部署拓扑图、分片键选择策略和RAM保存数据的重要性要格外小心，然后MongoDB分片集群才可以全心全意、竭尽所能为我所用。

部署与管理

Deployment and Administration

本章内容

- 配置与硬件需求
- 监控与诊断
- 备份与管理任务
- 安全
- 性能与错误解决
- 部署检查列表



本书如果不包含一些部署与管理的内容就不算完整。毕竟，使用MongoDB和保证它在生产环境中平稳运行是两码事。最后一章的目的是让大家在部署和管理MongoDB的时候可以做出正确的决策。我们可以把本章作为生产环境数据库部署的智慧经验。

本章首先介绍MongoDB的硬件需求和一些部署的参数，然后继续介绍一些保证系统运行、伸缩性和安全的方法。最后我们会介绍部署的检查列表供大家参考，确保部署已经完全符合要求。

13.1 硬件与配置

Hardware and provisioning

在部署MongoDB之前要问自己的第一个问题，“我应该部署到什么上面？”如果在单个笔记本电脑上运行整个生产集群，和我们本书前面做的一样，则可能会有大麻烦。本节里，我们将讨论如何为需求选择正确的拓扑，不同的硬件如何影响MongoDB，以及硬件配置的参数。

13.1.1 集群拓扑

本节会介绍集群拓扑的一些基本推荐配置，但是对于主从复制和分片集群的不同部署拓扑的

完整分析可以参考第11章和第12章。

图13.1所示为3种不同集群的最小的配置，以及可以升级到不同类型的集群的时机。

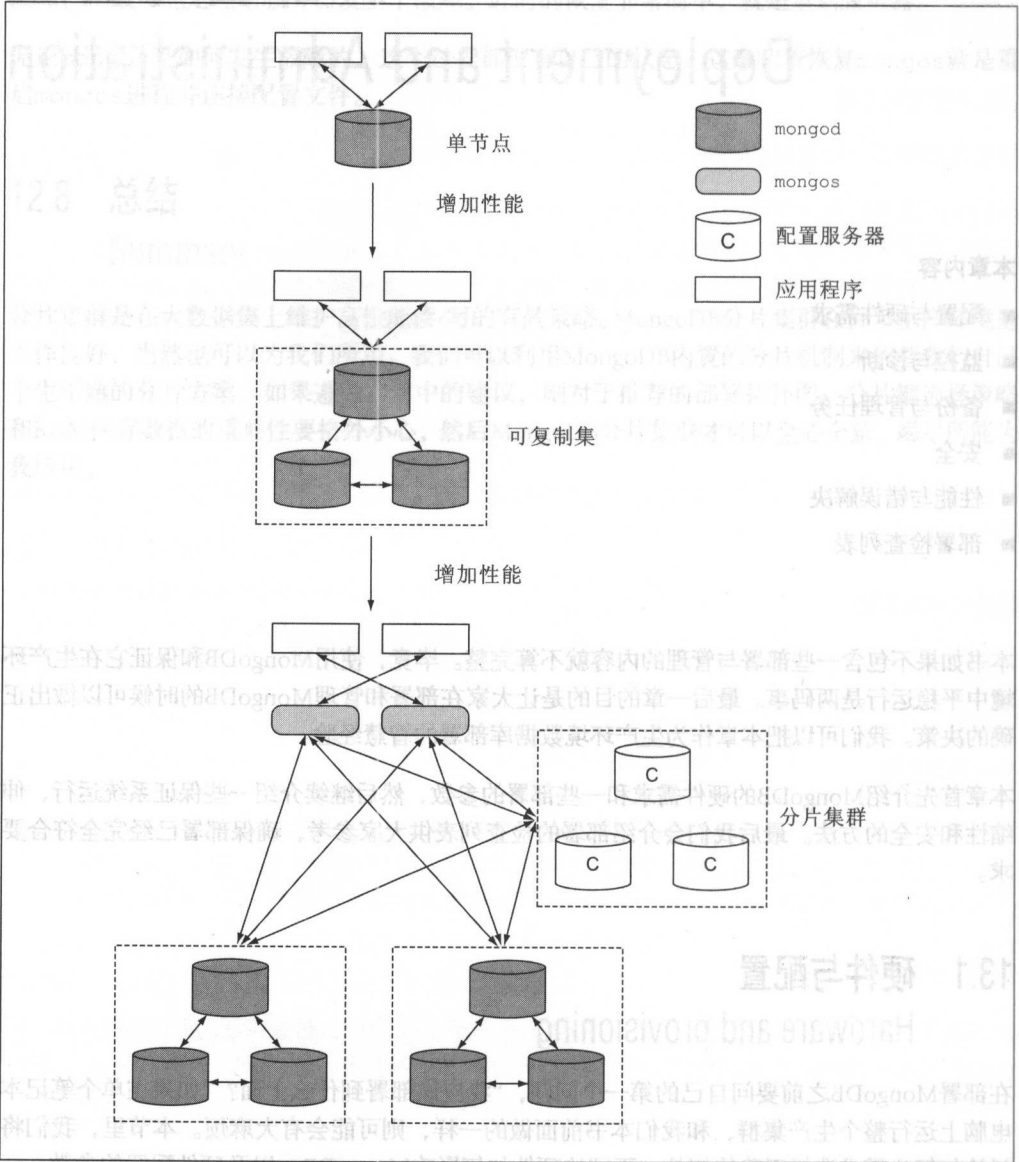


图 13.1 最小的单节点、可复制集和分片集群安装

MongoDB共有3种不同类型的集群：

- 单节点 正如我们在图13.1中所看到的，MongoDB可以在测试环境使用单服务器模式。但

是对于生产环境部署，不推荐单服务器模式，即使启用了日志。只有一台机器难以实现备份和恢复，而且当这台机器宕机时没有灾备的机器。也就是说，只有不需要可靠性和数据集比较小时，通常才会选择这个。

- 可复制集 如图13.1所示的，可复制集部署的最低推荐配置是3个节点，最少是2个数据存储节点和1个裁判节点。可复制集对于自动化灾备是必须的，更容易被封，而且不会有单点故障。更多关于可复制集的内容可以参考第10章。
- 分片集群 如图13.1底部所示，最小的分片集群配置有2个片，只有一个片时将会增加额外的压力且无法利用分片的优势。每个分片都应该是个可复制集，并且有3个配置服务器来确保没有单点故障。注意，还有2个mongos路由进程。丢失所有的mongos进程不会导致任何数据丢失，但是仍然会有恢复时间，所以我们用2个进程来保证生产环境下的高可用性。当想要通过捆绑廉价服务器来提升容量的时候可以使用分片集群。请参考第12章关于分片集群的更多详细信息。

既然我们已经完全理解了集群的类型，现在就来详细学习每个单独的服务器的部署细节。

13.1.2 部署环境

这里我们会介绍如何为MongoDB选择良好的部署环境。讨论硬件需求，比如CPU、RAM和磁盘，以及推荐的操作系统优化方案。图13.2是个简化示意图，包含了MongoDB要交互的硬件设备。在后续的章节里我们会讨论集群拓扑并提供在云中部署的一些建议。

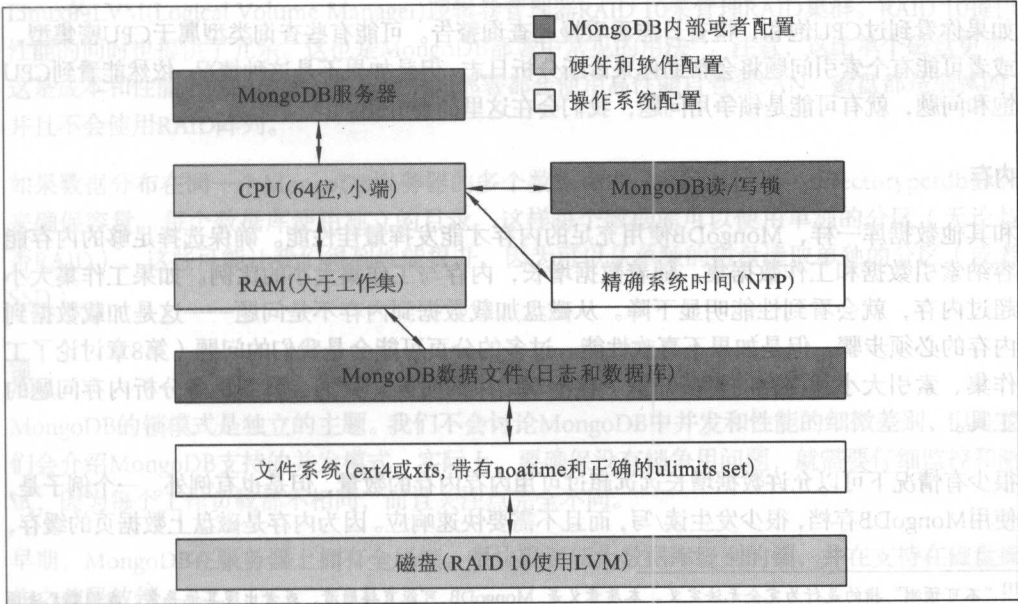


图 13.2 MongoDB 如何依赖操作系统和硬件的示意图

架构

硬件架构上的2个节点是有顺序的。首先，因为MongoDB映射所有的数据文件到虚拟地址空间上，所以所有的生产环境部署都应该在64位机器上。32位系统会限制MongoDB为2GB的存储空间。启用日志的时候，限制空间为大约1.5 GB。这在生产环境下十分危险，因为一旦超过限制，MongoDB的表现无法预测^[1]。但是可以在32位系统上进行单元测试，生产环境下的压力测试最好使用64位架构。

MongoDB服务器的组件必须运行在采取小端存储(little-endian)的机器上^[2]。这些不难兼容，因为x86基本都是使用这种方式，但是运行SPARC、PowerPC、PA-RISC等大端存储(big-endian)架构就无法使用^[3]。

客户端驱动虽然是分开维护的，但它们通常会支持小和大端字节顺序。这意味着虽然服务器必须运行在小端存储机器上，但MongoDB的客户端通常可以运行每个架构。

CPU

MongoDB并非CPU密集型，因为数据库操作很少是CPU密集型的，所以在诊断性能时这不是第一个要查看的地方。当优化MongoDB时，第一个工作就是要确保操作不是I/O密集型的（我们会在接下来两节讨论RAM和磁盘时介绍I/O问题）。

一旦索引和工作集完全填充进内存，就可能会看到更多的CPU消耗。如果一个MongoDB实例服务要每秒几万几十万查询，就需要更多的CPU内核才能提高性能。

如果你看到过CPU饱和，检查日志看看慢速查询警告。可能有些查询类型属于CPU密集型，或者可能有个索引问题将会帮助你来诊断分析日志。但是如果不是这种情况，依然能看到CPU饱和问题，就有可能是锁争用问题，我们会在这里简要介绍。

内存

和其他数据库一样，MongoDB使用充足的内存才能发挥最佳性能。确保选择足够的内存能容纳索引数据和工作数据集。随着数据增长，内存与工作集大小的比例。如果工作集大小超过内存，就会看到性能明显下降。从磁盘加载数据到内存不是问题——这是加载数据到内存的必须步骤。但是如果不喜欢性能，过多的分页可能会是我们的问题（第8章讨论了工作集、索引大小和内存的更多知识）在本章结束的时候，我们会看到更多分析内存问题的工具。

很少有情况下可以允许数据增长远远超过可用内存内存的数量，但是也有例外。一个例子是，使用MongoDB存档，很少发生读/写，而且不需要快速响应。因为内存是磁盘上数据页的缓存，

^[1] “不可预测”指的是行为完全无法定义。本质意义是 MongoDB 可能直接崩溃，或者出现其他异常。我们都无法预测。这里主要是强调我们应该避免这种场景，因为 MongoDB 的维护者也无法帮助你。

^[2] 机器的“端”是硬件知识，指的是内存中字节存储的顺序。参考 <https://en.wikipedia.org/wiki/Endianness>。

^[3] 如果你对于大端存储支持感兴趣，可以参考 <https://jira.mongodb.org/browse/SERVER-1625>。

如果在短期内使用次数超过一次，则它可以提供性能加速。在存档程序中，显然不符合这种情况，页面会在第一次使用的时候（可能只有一次）从磁盘加载到内存。为这种程序提供太多的内存可能是个浪费。对于所有的数据集，测试是关键。测试应用原型来确保获得最基本的性能。

磁盘

选择磁盘时，我们需要思考成本、IOPS（每秒输入/输出）、搜索时间和存储能力。就单个消费者级别磁盘、云端虚拟磁盘（或EBS）、高性能SAN之间的差别，这里不会过多介绍。一些程序在单网络附加EBS速度可以接受，但是高要求应用需要的不同。

磁盘性能非常重要，原因如下：

- 高写入工作负载——当数据写入MongoDB时，服务器必须把数据写入磁盘。对于高并发写入的APP和慢速磁盘，写入操作会影响整个系统的性能。
- 快速磁盘允许更快的服务器预热——任意时候重启服务器，都必须把数据集加载到内存里。这种情况很少发生。每个连续的MongoDB读或写都会加载新的虚拟内存到内存里，直到物理内存占满。快速磁盘会加速过程，显著提高MongoDB的性能。
- 快速磁盘会调整应用程序必须的工作集大小，达到内存的比率——使用SSD磁盘，可能需要更少的内存（或者更大的容量）。

无论使用什么类型的磁盘，正式部署都不会使用单个磁盘，而是使用RAID阵列。用户使用Linux的LVM(Logical Volume Manager)逻辑卷管理器RAID 10来管理RAID集群。RAID 10提供性能的同时也提供了冗余。这也是MongoDB部署中常见的用法^[1]。注意，这比单个磁盘更贵，这是成本和性能之间的权衡。更高级别的部署都会使用高性能自管理SAN，磁盘都是细腻的，并且不会使用RAID阵列。

如果数据分布在同一个MongoDB服务器的多个数据库中，就可以使用—directoryperdb标志来确保容量，每个数据库使用独立的目录。这样每个数据库可以使用单独的分区（无论是否RAID）。这些可能让我们得到性能提升，因为可以从单独的磁盘读取单独的分区（或者SSD）。

锁

MongoDB的锁模式是独立的主题。我们不会讨论MongoDB中并发和性能的细微差别，但是我们会介绍MongoDB支持的并发模式。实际上，要确保没有锁争用问题，就需要仔细监控和测试，因为每个工作负载都不相同，而且关注点完全不同。

早期，MongoDB在服务器上拥有全局锁。很快就更新为数据库级别的锁，并在支持在磁盘操作之前释放锁。

^[1] RAID 级别参考 http://en.wikipedia.org/wiki/Standard_RAID_levels。

MongoDB 3.0现在支持2个独立的存储引擎，支持不同的并发模式。它在基于mmap的原生存储引擎里包含集合级别的锁，WiredTiger存储引擎支持文档级别的锁，它可以替代原生的存储引擎。Consult JIRA、MongoDB的Bug跟踪系统包含这些改进的详细信息^[1]。

文件系统

如果在正确的文件系统上运行MongoDB，就会获得最佳的性能。有两个特别之处，ext4和xfs，功能快速、持续的磁盘分配。使用这个文件系统会加速MongoDB频繁的预分配操作^[2]。

安装快速文件系统后，就可以通过禁用更新文件的最后访问时间`atime`来再次提升性能。通常，无论读或者写的时候操作系统会更新文件的`atime`。在数据库环境中，这些都会带来许多不必要的工作。在Linux系统禁用`atime`比较简单：

(1) 备份配置文件：

```
sudo cp /etc/fstab /etc/fstab.bak
```

(2) 使用喜欢的编辑器打开最初的文件：

```
sudo vim /etc/fstab
```

(3) 对于每个安装的卷，可以在`/etc/fstab`找到，我们会看到根据柱面分类的设置列表。在下面添加`noatime`指令：

```
# file-system mount type options dump pass
UUID=8309beda-bf62-43 /ssd ext4 noatime 0 2
```

(4) 保存工作。新的设置会立即起作用^[3]。

可以使用`findmnt`命令来查看所有Linux系统支持的文件系统：

```
$ findmnt -s
TARGET SOURCE FSTYPE OPTIONS
/proc proc proc defaults
/ /dev/xvda ext3 noatime,errors=remount-ro
none /dev/xvdb swap sw
```

-s选项让`findmnt`命令从`/etc/fstab`文件获取数据。不带参数运行`findmnt`命令，会显示更多细节。

文件描述符

一些Linux系统允许打开的文件描述符的最大数量是1024。对于MongoDB而言有时候觉得太慢了，打开连接时可能导致警告信息或者错误（会在日志里看到）。自然而然，MongoDB需

^[1]可以在这里找到发布提示：<https://jira.mongodb.org/browse/server> 和 <https://docs.mongodb.org/manual/release-notes/>。

^[2]推荐设置参考：<https://docs.mongodb.org/manual/administration/production-notes/#kernel-and-file-systems>。

^[3]请注意这些是基本建议，但是并没有包含全部内容。参考：<https://en.wikipedia.org/wiki/Fstab>。

要为每个打开的文件和网络连接一个文件描述符。

假设在文件加里存储的数据文件包含了“data”单词，就可以使用`lsof`查看数据文件描述符的数量：

```
lsof | grep mongo | grep data | wc -l
```

查看网络连接描述符数量的代码，也比较简单：

```
lsof | grep mongo | grep TCP | wc -l
```

当处理文件描述符时，最佳的策略是从一个高的限制开始，这样就不会在生产环境中耗尽资源。也可以使用`ulimit`命令来检查当前的临时限制：

```
ulimit -Hn
```

要永久启用限制，就要在选择的编辑器里打开`limits.conf`文件：

```
sudo vim /etc/security/limits.conf
```

然后设置软性和硬性限制。这是基于每个用户的。这个例子假设`mongodb`用户运行`mongod`进程。

```
mongodb soft nofile 2048
mongodb hard nofile 10240
```

这些新的设置将会在用户重新登录时起作用^[1]。

时钟

事实证明，复制会受到“时钟偏差的影响”，这个问题会在主节点和从节点服务器的时间不同时出现。复制依赖于时间比，所以如果同一个复制集群中的机器时间不同，可能会导致严重的问题。这不是所期望的，幸亏有解决办法。我们最好确保每个服务器使用NTP（网络时间协议）^[2]或者某些时间同步协议来保证时间一致：

- 在UNIX上，运行`ntpd`。
- 在Windows上，Windows时间服务负责这项工作。

日志

MongoDB 1.8引入了日志功能。从版本2.0开始，MongoDB默认启用了日志，它会在写入核心数据文件之前先写入日志。

这允许MongoDB服务器在意外关机事故中快速恢复在线。

^[1]如果用户只进行纯守护，则可能不起作用。此时，使用`sudo service mongodb restart`命令重启`mongodb`服务可解决问题。

^[2]使用类似`ntpstart`命令检查NTP，确保其工作正常。

如果没有启用日志的mongod进程意外关闭，恢复一致性数据文件就需要运行修复。修改进程会重写数据文件，抛弃不理解的数据（冲突的数据）。因为停机时间和数据丢失通常令人痛苦，这种修复方式通常是最后的办法。从现有的节点同步数据的方式更加简单和可靠。这也是为什么要使用主从复制的原因。

日志减轻了数据库修复的必要，因为MongoDB可以使用日志来恢复数据文件到一致状态。在MongoDB 2.0和MongoDB 3.0里，日志是默认启动的。我们可以使用--nojournal参数来禁用日志：

```
$ mongod --nojournal
```

当启用的时候，日志文件会保存在叫journal的目录中，在主数据文件的目录中。

如果启用日志来运行MongoDB服务器，记住下面的要点：

- 日志需要额外的写操作。
- 解决这个问题的一个办法是独立分配日志磁盘，然后创建一个符号链接（symlink）^[1]，指定日志文件路径。分区容量尽量大，到120 GB足够了，这种容量的SSD价格可以接受。单独SSD存放日志文件，确保影响降到最低。
- 日志无法担保不丢失写入数据。它只保证MongoDB恢复时的一致性状态。

日志每隔100 ms会同步日志到磁盘，所以意外关闭可能会丢失100 ms中的写入数据。如果业务逻辑无法接受，则可以通过任意客户端驱动来修改写关注。可以安全模式运行（比如w和wtimeout）。例如，在Ruby驱动里，可以使用j选项让服务器永远运行在安全模式：

```
client = Mongo::Client.new( ['127.0.0.1:27017'], :write => {:j => true}, :database => 'garden')
```

要知道这样配置是不明智的，因为这会强制每个写入都必须等待下一个日志同步^[2]。甚至这些写入都需要回滚，因为日志值会等待主节点写入而不会等待从节点写入。因此，常见的使用方式是确保写入是持久化的，确保写入多数可复制服务器，这样通常足以应对失败。参考第10章关于复制和写入关注点选项的更多内容^[3]。

13.1.3 配置

我们可以部署到自己的硬件上，这个方法有自己的优势，但是这一小节里我们更多关注在云部署和自动化上。私有云部署需要许多专业的管理和安装，这些超出了本书的范围。

^[1]符号链接，本质上是个对象，看起来像文件，但是实际上指向了另外的地址。这里指的是 MongoDB 仍然可以在相同位置找到日志，虽然实际上它在完全不同的磁盘上。更多细节参考 https://en.wikipedia.org/wiki/Symbolic_link。

^[2]最近版本的 MongoDB 允许使用 commitIntervalMs 参数修改日志提交间隔时间，减少日志确认写入的延迟。

^[3]写关注级别的更多行为信息，参考 <https://docs.mongodb.org/manual/core/write-concern>。

越来越多的用户在虚拟化环境，也就是云中托管MongoDB。Amazon的EC2就是其中之一，因为使用简单，具有更多的数据中心和竞争性的价格。EC2和其他环境一样，适合部署MongoDB。更高级别上，当部署到云平台时有三个组件需要考虑：

- 谁来托管运行MongoDB？
- 持久化存储保存MongoDB文件的系统；
- MongoDB内部使用和与客户端通信的网络。

首先，EC2非常方便，因为易于配置，而且按需分配，所以成本会随着需求伸缩。但是，EC2的一个优势它是黑盒的。我们可能经历过服务高峰或者低谷，但是没有办法分析和诊断它们。有时候会获得一个“黑盒”，这意味着虚拟实例分配到慢速硬件服务器上或者还有其他活跃用户^[1]。

其次，EC2允许我们安装虚拟块服务，如EBS，作为永久存储系统。EBS提供了更大的灵活性，我们可以根据需要添加和删除容量。EBS也支持快照，可以用来备份。

使用最便宜的EBS存储的问题是它们无法提供像物理磁盘一样的高级别的吞吐量。通过确保核实主机提供商来了解准确的信息。对于写入，EBS可能是SSD，Amazon也提供了选项，另外一种增加性能的方式是使用RAID 10来增加读吞吐量。

最后，EC2网络对于大部分用户来说已经足够了，但是像存储和主机，通过使用EC2的网络会放弃许多控制。因为与其他主机共享网络，所以其他程序的高流量会影响网络的性能。像磁盘和实例类型，这些都是付费的，所以在解决问题的时候要注意这些细节。

综上所述，部署在EC2上有许多优势，但是显然也有许多缺点。因此，许多用户不想受制于EC2，选择了自己的物理设备运行MongoDB。

再强调一次，EC2和其他云非常方便，并且广泛应用于各种用户。最好的部署方法就是仔细测试。测试你要部署的平台，可以帮助我们做出决策。

MMS 自动化

另外一个相对新的配置选项是MongoDB惯例系统自动化（MMS）。MMS自动化可以配置EC2的实例，并一键安装整个集群^[2]。MongoDB的MMS团队通过不断添加新特性来简化MongoDB运营，所以如果你对此感兴趣，就可以参考最新的文档<https://docs.mms.mongodb.com>，获取最新信息。

^[1]注意 EC2 有许多不同的实例类型和购买选项。检查 EC2 或者其他云计算提供商的准确配置。

^[2]如果没有使用 EC2 虚拟主机，MMS Automation 也可以部署到预先配置的硬件上。

13.2 监控与诊断

Monitoring and diagnostics

一旦在生产环境里部署了MongoDB，就想要实时了解它的状态。一旦性能降低，或者频繁故障，就得通知用户。这就是监控的作用。

我们先从最简单的监控开始：日志。然后介绍内置的查看运行MongoDB服务器的状态命令。这些命令背后是mongostat工具和Web控制台，我们会简要介绍它们。

接下来会看一下MongoDB公司提供的MMS监控工具，也会推荐一些外部的监控工具。最后介绍2个诊断工具：bsondump和mongosniff。

13.2.1 日志

日志是监控的第一个级别，我们应该计划为所有的部署启用日志。这通常不是问题，因为MongoDB在启动时可以使用`-logpath`指定日志的路径。但是还有一些其他的设置需要了解，比如，要启用详细日志，就要在启动mongod的时候使用`-vvvvv`参数（v越多输出信息越详细）。例如，调试代码并且记录每个查询，这非常方便。但是要知道详细日志模式就会使得保存的日志信息变得更大而且影响服务器性能。

如果日志变得非常大，就在不同的分区里存储日志文件。

接下来我们可以使用`-logappend`启动mongod。这会追加日志到现有的日志，而不是默认模式，使用时间戳来滚动文件。

当有个长期运行的MongoDB进程时，你可能想自己写个脚本来定期滚动日志文件。其实不需要，MongoDB已经提供了logrotate来完成这个操作。可以直接从shell里启动使用：

```
> use admin
> db.runCommand({logrotate: 1})
```

发送SIGUSR1^[1]信号给进程，并且运行logrotate命令：

这是向进程12345发送信号的代码：

```
$ kill -SIGUSR1 12345
```

我们也可以使用ps命令来查找进程的ID，如下所示：

```
$ ps -ef | grep mongo
```

注意，kill命令没有听起来这样可怕。它只是给进程发送信号，但是绝大部分操作都是结束进

^[1]类 UNIX 系统支持“signals”发送信号给运行的进程，这样可以在进程接受到信号后触发某个操作。MongoDB 可以接受 SIGUSR1 信号并且滚动日志文件。

程，因此而得名^[1]。但是，使用`-9`参数来运行`kill`，就会使用粗暴的方式结束进程。这在生产环境下应尽量避免。

13.2.2 诊断命令

MongoDB包含了许多命令可以报告内部状态。这些命令背后都是监控MongoDB。这里是一些常见命令的参考说明，应该对大家有点帮助：

- 全局服务器统计信息: `db.serverStatus()`;
- 统计当前运行的操作: `db.currentOp()`;
- 包含空闲的操作: `db.currentOp(true)`。
- 每个数据库的计数器和活动统计: `db.runCommand({top:1})`;
- 内存和磁盘的使用情况统计: `db.stats()`。

每个MongoDB版本都会改进这些命令的输出结果，所以本书中介绍的信息可能过时。请参考自己的MongoDB对应的版本说明文档。

13.2.3 诊断工具

除了前面列举的诊断命令，MongoDB还提供了一些诊断工具。大部分都是基于之前的命令构建的，而且可以使用驱动或者shell轻易调用。

以下是我们本节将要介绍的内容：

- `mongostat`——全局系统统计信息；
- `mongotop`——全局操作统计信息；
- `mongosniff` (高级)——监控MongoDB网络流量；
- `bsondump`——以JSON格式显示BSON文件。

MONGOSTAT

用`db.currentOp()`方法只显示某个时刻队列化的操作或者正在处理的操作。类似地，`serverStatus`提供了系统某个时间点的各种信息统计和计数器。但是有时候，我们想看一下系统的实时行为，这是`mongostat`的工作。根据`iostat`和其他相似的工具建模，`mongostat`定时轮训并显示统计从每秒插入的数量到内存的使用量，再到B-树页缺失的频率等信息。

我们也可以本地调用`mongostat`命令，这会每秒轮训一次：

^[1]Windows 系统还没有等价工具，所以必须使用 `logrotate` 命令。

```
$ mongostat
```

它也具备良好的配置性，我们可以使用`-help`参数来查看它所有的参数。例如，可以使用`-host`参数来连接特别的主机和端口，而不是默认的`localhost:27017`。其中一个重要的特性就是集群发现。当使用`-discover`参数启动`mongostat`时，也可以使用`-host`参数来指定监控的节点服务器，并且也可以发现可复制集群或者分片集群中的节点。然后会显示整个集群的统计信息。

mongotop

`mongostat`是`db.currentOp()`和`serverStatus`命令的外部工具，与之类似的`mongotop`是`top`命令的外部包装工具，其运行方式与`mongostat`的一样，假设我们已经有个MongoDB数据库默认端口运行在本机了。

```
$ mongotop
```

与`mongostat`类似，我们也可以使用`-help`参数运行这个命令，查看可用的配置参数。

mongosniff

接下来我们将介绍`mongosniff`工具，它可以侦探客户端发送给MongoDB服务器的请求包并打印出来。如果你恰巧要编写一个驱动或者调试程序，这就是要找的工具。我们可以用如下代码来启动它，侦听默认的地址和端口：

```
sudo mongosniff --source NET IO
```

然后，当使用任意客户端连接时——假设是MongoDB shell——我们会得到一串字符串信息：

```
127.0.0.1:58022 -->> 127.0.0.1:27017 test.$cmd 61 bytes id:89ac9c1d
2309790749 query: { isMaster: 1.0 } ntoreturn: -1
127.0.0.1:27017 <--- 127.0.0.1:58022 87 bytes
reply n:1 cursorId: 0 { ismaster: true, ok: 1.0 }
```

从这里我们看出，客户端运行了`isMaster`命令，表示在`test.$cmd`集合上执行`{ isMaster: 1.0 }`查询。我们还看到了应答消息包含`ismaster: true`文档，这表示此节点是主节点。使用`-help`运行`mongosniff`可以查看所有参数。

bsondump

另外一个有用的命令就是`bsondump`，它允许我们检查原始的BSON文件。BSON文件由`mongodump`命令工具生成（13.3小节讨论）并且通过可复制集回滚^[1]。例如，假设我们已经把一个包的集合导出到一个文档里，集合的文件是`users.bson`，那么可以用以下代码很方便地检查内容：

```
$ bsondump users.bson
{ "_id" : ObjectId( "4d82836dc3efdb9915012b91" ), "name" : "Kyle" }
```

^[1] MongoDB把可复制集回滚文件写入数据目录下的回滚目录中。

正如我们看到的，bsondump使用JSON格式打印了BSON的数据。如果使用严格的调试过程，就可以看到真实的BSON类型的组成结构和大小。

```
$ bsondump --type=debug users.bson
--- new object ---
size : 37
  _id
type: 7 size: 17
  name
type: 2 size: 15
```

这里显示了对象的总大小(37B)，2个字段的类型(7 和2)和字段大小。

Web 控制台

最后，MongoDB通过Web接口和REST服务器提供了一些访问统计信息的方式。以其3.0版本为例，这些系统已经旧了，不再开发。但是与早期其他工具或者数据库命令输出的信息是一样的。如果要使用这些系统，请确保阅读最新的文档并自习考虑安全问题。

13.2.4 监控服务

MongoDB公司提供了免费的MMS监控工具，不仅提供了控制面板，还提供了分享系统信息的功能，这些有助于了解系统。MMS监控也可以作为大企业付费单独运行版本授权管理。若需要使用，就要在MMS Monitoring官方网站<https://mms.mongodb.com>注册个账号。一旦注册完成，就会看到如何使用MMS监控工具的步骤。

13.2.5 外部监控应用

绝大部分部署都需要一个外部监控应用。Nagios和Munin是两个非常流行的开源监控工具，可以帮助我们实时监控MongoDB数据库。简单安装插件即可使用。

为任意的监控程序编写插件并不困难。本质上都是在MongoDB数据库中运行统计命令。通常serverStatus、dbstats、collstats这三个命令提供了全部所需信息。我们也可以直接从HTTP REST接口获取信息，避免使用驱动程序。

最后，不要忘记低级别的系统监控工具的价值。例如，iostat命令可以帮助诊断MongoDB的性能。MongoDB部署的绝大部分性能问题都可以追本溯源到一个根源上：磁盘。

下面的例子中，我们使用了-x参数来显示扩展统计信息，其中“2”表示每隔2秒显示一次：

```
$ iostat -x 2
Device: rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sdb 0.00 3101.12 10.09 32.83 101.39 1.34 29.36
Device: rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sdb 0.00 2933.93 9.87 23.72 125.23 1.47 34.13
```


对于每个字段的详细描述信息，或者特定版本的iostat信息，可以查看官方主页^[1]。对于快速诊断，主要感兴趣的信息是下面2列：

- await列表示I/O请求处理的平均时间（毫秒）。这个平均时间包含I/O队列和服务I/O请求的时间。
- %util是CPU的百分比，本质上也就是设备的使用带宽。

前面的iostat片段显示了磁盘的使用情况。磁盘I/O的平均等待时间是100 ms（提示：太多了），使用率是30%。如果分析MongoDB日志文件，就会看到许多的慢操作（查询、插入等）。事实上，这些慢速的操作可能提示我们有潜在的问题。我们已经介绍了多种诊断和监控正在运行的MongoDB服务的方法，用来保证它平滑运行。下面我们来看看生产环境下不可避免的问题：备份。

13.3 备份

Backups

生产环境部署数据库的一个考虑就是要应对灾难。备份在这种过程中扮演着重要角色。当出现故障时，良好的备份可以节约大量恢复的时间，此时，我们更不会后悔花时间去执行常规的备份。虽然仍然有很多用户认为无需备份也可以活得很好，但当他们不能恢复数据库的时候，只有责怪自己了。

对于备份MongoDB数据库，有一些通用的规则，如下所示：

- 使用mongodump和mongorestore；
- 复制原始数据文件；
- 使用MMS备份。

接下来的三节内容我们会学习这些策略。

13.3.1 mongodump 和 mongorestore

mongodump用于把数据库内容写为BSON文件。mongorestore用于读取这些文件并恢复它们。这些工具对于备份单个集合和整个数据库都十分有用。它们也可以用于备份实时运行的数据库（无需锁住或者关闭数据库），或者把它们指向到一个数据文件的集合上，

但是只有当服务器被锁住或者关闭时。mongodump最简单的运行方式如下所示^[2]：

^[1]手册页面，是基于 Unix 的系统提供文档的方式。例如，在终端里输入 iostat 来获取其帮助信息。

^[2]如果启用从节点读功能，可以在可复制集的从节点上使用此功能。

```
$ mongodump -h localhost --port 27017
```

它会把localhost服务器上的每个数据库和集合备份到dump^[1]目录里。Dump目录包含每个集合的所有文档，包含定义用户和索引的系统集合。但是非常明显，索引本身的数据不会包含到dump目录里。这意味着恢复数据库时索引需要重建。如果有很大的数据集，或者许多索引，则会花费很多时间。

恢复 BSON 文件

要恢复BSON文件，就运行mongorestore命令，指向dump转储文件夹即可：

```
$ mongorestore -h localhost --port 27017 dump
```

注意，当恢复数据时，mongorestore不会删除数据，所以如果恢复的是存在的数据库，就确保使用-drop标志运行。

13.3.2 基于数据文件的备份

绝大部分用户选择基于文件的备份方式，直接把原始的数据文件拷贝到别的地方。这个方法比使用mongodump快得多，因为备份和恢复不需要转换数据格式。

这种方式唯一的问题就是它需要锁住数据库服务器，但是通常只是需要锁住从节点，因此应该可以在备份过程中保证应用的正常运行。

复制数据文件

用户经常犯错，没有锁住数据库的时候就复制数据文件。虽然启用日志，这样做也会导致复制文件中断。本节里将会介绍如何通过锁住数据库来允许安全备份，保证数据文件的一致性状态。

实时系统快照

你可能好奇，为什么启用日志以后还必须锁住数据文件？答案是，日志只能从某个时间点把数据库恢复到一致的状态。如果手动复制数据文件，则在复制每个文件的时候还是有延迟存在，这意味着不同的数据文件可能对应不同的时间点，并且日志无法处理这个问题。

但是如果文件系统、存储提供者或者托管提供其明确支持时间点快照，就可以使用这个功能来确保安全的快照，而不需要锁住数据文件。注意，每个东西都会在相同的时间里保存，也包括日志。这意味着日志在不同的磁盘上（或者已经禁用日志），除非系统支持跨多卷的时间点快照，否则比较麻烦。

^[1] 如果可以，则尽量存储此目录到不同的磁盘上，这是为了提高性能和防止磁盘失败。

为了安全地拷贝数据文件，首先需要确保它们状态的一致性，所以要么关闭数据库要么锁住它。因为关闭数据库可能涉及一些部署工作，所以绝大部分用户选择锁住数据库。以下是同步和锁住的命令：

```
> use admin
> db.fsyncLock()
```

此刻，基于写入锁住数据库^[1]并且数据文件同步到磁盘。这意味着现在可以安全地拷贝数据文件了。如果正在支持快照的文件系统或者存储系统上运行，最好接下来就获取快照并复制数据。这样允许我们快速解锁。

如果你不能运行快照，就必须在复制数据文件的时候锁住数据库。如果从从节点复制数据文件，就要确保从节点与主节点有足够的oplog来保存离线备份期间的操作日志。

一旦完全快照或者备份，就可以解锁数据库了。解锁数据库的命令如下所示：

```
> db.fsyncUnlock()
```

注意，这仅仅是个解锁请求，数据库可能无法正确解锁。运行db.currentOp()方法可以验证数据库是否还处于锁定状态。

13.3.3 MMS 备份

再说一遍，MongoDB的MMS团队也提供了一个解决方案。MMS备份使用了操作日志来提供某一时间点的整个集群备份。它可以使用于单个可复制集群也可以用于整个分片集群中。正如我们之前提到的，MMS团队一直在添加新功能，所以请大家使用的时候参考最新的官方文档。

13.4 安全

Security

安全是极其重要但是又容易在生产环境部署数据库时忽略的问题。本节里我们将会介绍主要的安全类型，包含环境安全、网络加密、验证和授权。

最后，我们会简要介绍MongoDB企业版支持的安全特性。其重要性可能超过了其他主题，了解最新的安全工具和最佳实践至关重要，所以可以把本节作为安全设计的参考总览。但是在生产环境使用的时候请参考最新的官方文档<https://docs.mongodb.org/manual/security>。

^[1]锁之后的任意尝试写入都会被阻止，而且读操作也会被阻止。很不幸也包含验证数据，这意味着新的连接尝试也会被阻止，所以记住当运行备份的时候保证连接是打开的。参考官方文档 <https://docs.mongodb.org/manual/reference/method/db.fsycnLock/> 获取更多信息。

13.4.1 安全环境

与其他的数据库类似，MongoDB应该运行在安全的环境中。MongoDB生产环境的用户必须利用现代操作系统的安全特性来保证数据的安全。其中最重要的方式之一就是防火墙。

与MongoDB一起使用防火墙的唯一潜在困难点就是需要知道哪些机器需要彼此通信。幸运的是，这些通信的规则非常简单：

- 使用可复制集时，每个节点必须能与其他节点通信。
- 所有的数据库客户端必须能够与每个可复制集群的节点通信。
- 所有的通信都使用TCP协议。
- 对于可达节点，指的是它可以通过配置的端口进行通信连接。例如，mongod侦听在TCP端口27017上，所以，可达指的是它可以通过此端口连接。

分片集群由可复制集组成。所有可复制集的应用规则：分片集群的客户端是mongos路由器。此外：

- 所有的分片必须可以与其他节点直接通信。
- 分片和mongos路由必须能够与配置服务器通信。

图13.2为这些连接规则的简单视图，任意指向可复制集盒子的或者配置服务器的箭头，表示连接到盒子里的每个服务器。

大部分情况下，在安全环境里运行MongoDB是独立的主题，但是`--bind_ip`与运行相关^[1]。默认情况下，MongoDB会侦听机器上的所有地址，但是可能我们想侦听某个端口。为此，可以使用`--bind_ip`参数来启动mongod和mongos，它可以自定多个IP地址，用逗号分隔。例如，只要侦听在外部端口上，可以如下来启动mongod：

```
mongod --bind_ip 127.0.0.1,10.4.1.55
```

注意，发送的数据是明文的，除非启用SSL。我们会在下一节里介绍SSL。

13.4.2 网络安全

保护系统最根本的方面可能就是确保网络的安全。即使系统是完全孤立的，并且没有人可以查看数据（例如，如果所有的通信都在vpn里加密了，或者路由规则不允许外部非信任的网络发送数据到我们的机器^[2]），我们也应该对MongoDB加密。幸运的是，MongoDB 2.4发布了处理加密的库——安全套接层(SSL)——内置提供了支持。

^[1]注意，对于2.6版本，Linux预编译包默认包含`--bind_ip`。

^[2]许多云托管服务提供了帮助工具，比如VPCs、子网和AWS中的安全组。

要看一下为什么它如此重要。假设我们使用UNIX命令tcpdump窃听消息。首先使用ifconfig命令查看程序通信使用的回环接口。

下面是开始的输出结果:

```
$ ifconfig
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    :
    :
```

这里,回环接口就是lo。现在我们可以使用tcpdump来转储此端口的网络通信数据:

```
$ sudo tcpdump -i lo -X
```

注意:使用tcpdump读取网络流量数据需要root权限,所以如果你不能运行此命令,就使用例子的参考命令操作。

在同一个机器的另外一个终端里启动mongod,但是不启用SSL(修改恰当的数据路径):

```
$ mongod --dbpath /data/db/
```

然后,连接数据库,插入单个文档:

```
$ mongo
>
> db.test.insert({ "message" : "plaintext" })
> exit
bye
```

现在,如果在终端里查看tcpdump输出,就会看到许多输出包,如下所示:

```
16:05:10.507867 IP localhost.localdomain.50891 >
  → localhost.localdomain.27017 ...
0x0000: 4500 007f aa4a 4000 4006 922c 7f00 0001 E....J@.@.,...
0x0010: 7f00 0001 c6cb 6989 cf17 1d67 d7e6 c88f .....i....g....
0x0020: 8018 0156 fe73 0000 0101 080a 0018 062e ...V.s.....
0x0030: 0017 b6f6 4b00 0000 0300 0000 ffff ffff ....K.....
0x0040: d207 0000 0000 0000 7465 7374 2e74 6573 .....test.tes
0x0050: 7400 2d00 0000 075f 6964 0054 7f7b 0649 t.-...._id.T.{.I
0x0060: 45fa 2cfc 65c5 8402 6d65 7373 6167 6500 E.,e...message.
0x0070: 0a00 0000 706c 6169 6e74 6578 7400 00 ...plaintext..
```

① 文档在网络
中明文传输

我们保存的信息就在最后2行①的位置明文显示!这也显示了网络加密多么重要。现在,我们来试验SSL运行MongoDB,看看会发生什么。

使用 SSL 运行 MongoDB

首先,要为服务器生成一个key密钥:

```
openssl req -newkey rsa:2048 -new -x509 -days 365 -nodes -out mongodbcert.
```

```
crt -keyout mongodb-cert.key
cat mongodb-cert.key mongodb-cert.crt > mongodb.pem
```

然后，试验SSL启动mongodb服务器，使用—sslPEMKeyFile和—sslMode两个参数：

```
$ mongod --sslMode requireSSL --sslPEMKeyFile mongodb.pem
```

现在，使用shell SSL连接数据库，进行相同的操作：

```
$ mongo --ssl
> db.test.insert({ "message" : "plaintext" })
> exit
bye
```

如果使用tcpdump回顾输出窗口里的信息，就会发现消息完全无法理解：

```
16:09:26.269944 IP localhost.localdomain.50899 >
localhost.localdomain.27017: ...
0x0000: 4500 009c 52c3 4000 4006 e996 7f00 0001 E...R.@.....
0x0010: 7f00 0001 c6d3 6989 c46a 4267 7ac5 5202 .....i..jBgZ.R.
0x0020: 8018 0173 fe90 0000 0101 080a 001b ed40 ...s.....@
0x0030: 001b 6c4c 1703 0300 637d b671 2e7b 499d ...lL....c}.q.{I.
0x0040: 3fe8 b303 2933 d04b ff5c 3ccf fac2 023d ?...)3.K.\<....=
0x0050: b2a1 28a0 6d3f f215 54ea 4396 7f55 f8de ..(.m?...T.C..U..
0x0060: bb8d 2e20 0889 f3db 2229 1645 ceed 2d20 .....")..E...-
0x0070: 1593 e508 6b33 9ae1 edb5 f099 9801 55ae ....k3.....U.
0x0080: d443 6a65 2345 019f 3121 c570 3d9d 31b4 .Cje#E..l!.p=.1.
0x0090: bf80 ea12 e7ca 8c4e 777a 45dd .....NwzE.
```

文档在网络中加密传输

成功了！现在我们的系统安全没有？不完全。恰当的加密只是安全的一个方面。下一节里我们会介绍如何鉴别用户的身份标识，并且会介绍如何针对每个用户进行细粒度的权限控制。

集群中的SSL

现在，我们已经知道了如何在mongo shell和mongod之间建立SSL连接。很明显，问题来了，如何扩展到整合集群上？幸运的是，这与之前的例子非常相似。在集群中使用--sslMode requireSSL参数启动每一个节点。如果没有使用SSL运行集群，就需要在升级的过程中确保不丢失连接。参考官方文档<https://docs.mongodb.org/manual/tutorial/upgrade-cluster-to-ssl/>。

保管好密钥

本章中的绝大部分安全机制依赖于密钥交换。要确保密钥安全（使用适当的权限存储），除需要的时候以外，不要在服务器之间分享。

一个极端的例子：如果在所有的机器上使用相同的密钥，这就意味着黑客只需要破解单个密钥就可以读取所有的网络通信数据。

如果不确定是否应该分享密钥，就查看MongoDB官方文档或者底层机制。

13.4.3 验证

安全的下面一层就是验证。什么是好的网络加密？如果任意网络上的用户都可以伪装成合法的用户并且可以做任意想做的事情呢？

验证允许我们使用安全的方式来验证用户的身份。首先，我们会讨论为什么要验证用户。然后会简要讨论这些概念如何应用到可复制集合分片集群中。一如既往，我们将会在这介绍核心概念，但是我们只是针对自己的MongoDB版本查询最新的官方文档以确保信息的及时性（<https://docs.mongodb.org/manual/core/authentication>）。

服务验证

验证的第一步是检验另一端的连接是否是可信任的。为什么这很重要？主要是为了防止中间人攻击（man-in-the-middle attack），中间人伪装成客户端和服务端来窃取数据。图13.3是简单的中间人攻击示意图。



图 13.3 中间人攻击

正如在图中看到的，中间人攻击正如名字的含义一样：

- 恶意攻击者伪装成服务器，与客户端建立连接，然后伪装成客户端与服务器建立连接。
- 它不能仅解密或者加密客户端与服务器之间的通信数据，但是它可以发送任意的消息给客户端和服务端。

幸运的是，还有希望！MongoDB 使用的SSL库不仅提供了加密机制，还提供了证书验证，它使用信任的但是未参与通信的第三方验证主体来检验发送密钥方的身份。理论上，攻击者没有控制第三方验证主体。

生成证书，并且由信任的第三方签名，这个叫做证书颁发机构（CA）。其概念超出本书的范围。有许多参数，每个主题都是独立的。先研究一下CA，比如Symantec、Comodo SSL、GlobalSign等，或者可以使用工具来简化过程，比如SSLMate工具。

一旦有了证书，就可以在MongoDB使用它了，如下所示：

```
mongod --clusterAuthMode x509 --sslMode requireSSL --sslPEMKeyFile server.pem
--sslCAFile ca.pem
mongo --ssl --sslPEMKeyFile client.pem
```

ca.pem包含了CA中的根证书链，client.pem由CA签名。服务器会使用ca.pem的内容来检验client.pem是否被CA签名并且信任它。

这些步骤可以确保没有恶意的程序可以和我们的数据库建立连接。接下来一节，我们会看到如何在单个数据库里对每个用户进行细粒度的权限控制。

用户验证

虽然服务验证可以阻止攻击者创建连接，但有时候我们想要系统授权或者取消单个用户的权限。

注意：从版本2.4到版本2.6,MangoDB的API改变很大，而且可能以后还会修改，请确保在设置的时候查询最新的文档。这里的例子是针对版本2.6 和版本3.0的，它支持基于角色的验证方式。

在MongoDB中，角色本质上是权限的集合，权限是可以在MongoDB里执行的操作。角色的概念非常有用，因为有时候角色的逻辑思想不能映射到基元数据库操作中。例如，内置的读角色不允许用户执行查询操作；它允许用户运行某些命令来显示有读权限的数据库和集合统计信息。

MongoDB内置了一些方便的角色，而且也支持用户自定义角色。这里就不详细介绍这些角色，如果需要自定义角色，我们推荐查询最新版本的官方文档。现在来看一下如何实战设置基本的mongod验证。

设置基本验证

首先，我们应该使用auth启动mongod节点。注意，如果这个节点在分片集群或者可复制集中，就需要通过传递参数来允许验证其他服务器。但是对于单节点来说，启用数据库就只需要一个标志：

```
$ mongod --auth
```

现在，第一次连接服务器，我们要添加管理员账号：

```
> use admin
> db.createUser(
  {
    user: "boss",
    pwd: "supersecretpassword",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
  }
)
```

在我们例子里，为用户账户设置了userAdminAnyDatabase角色，以允许用户完全访问系统，包括添加和删除用户，还有修改用户的权限。这是MongoDB的超级用户。

现在我们已经创建了管理员账号，就可以使用它来登录了：

```
> use admin
> db.auth("boss", "supersecretpassword")
```

下面我们为单个的数据库创建用户。我们再次使用createUser方法。主要的不同就是角色：

```
> use stocks
> db.createUser(
  {
    user: "trader",
    pwd: "youlikemoneytoo",
    roles: [ { role: "readWrite", db: "stocks" } ]
  }
)
> db.createUser(
  {
    user: "read-only-trader",
    pwd: "weshouldtotallyhangout",
    roles: [ { role: "read", db: "stocks" } ]
  }
)
```

现在trader用户在stocks数据库有readWrite角色，而read-only-trader角色只要read的权限。第一个用户可以读写stocks数据库，而第二个账户只能读数据。注意，因为我们把这些用户添加到stocks数据库，所以需要数据库进行验证：

```
> use stocks
> db.auth("trader", "youlikemoneytoo")
```

删除用户

要删除用户，可以使用dropUser帮助方法：

```
> use stocks
> db.dropUser("trader")
```

此方法在此有点大材小用，可以只使用revokeRolesFromUser帮助方法从系统中取消权限，而不是删除账号，并且可以使用grantRolesToUser帮助方法再次授权。

要关闭会话，不需要显示退出，用中断连接(关闭shell)的方式也可以实现。而且也有一个退出的帮助方法：

```
> db.logout()
```

自然而然地，我们也可以在驱动代码里使用这里介绍的安全机制。可以查看驱动API的详细信息。

本地例外

你可能已经注意到了，我们在验证连接之前能添加用户。这可以看成是一个安全漏洞，但是这也可以看成是MongoDB提供的一个便捷之处，叫做本地例外。

这意味着，如果服务器没有配置，任意本地机器的连接就可以获取完全的权限。正如我们期望的，在添加第一个用户后，未验证的连接就没有权限了，可以在命令行里传递 `--setParameter enableLocalhostAuthBypass=0` 来禁用这个行为，并且通过在第一次启动服务器里禁用验证来设置，添加用户，然后重启的时候启用验证。

这个方法只能达到这种安全——在本地例外的窗口里，任何人都可以进入系统——但是它是另外一种选择。

13.4.4 可复制集验证

可复制集支持相同的验证API，但是为可复制集启用验证需要额外的配置，因为不仅客户端需要验证可复制集，而且可复制集节点也需要可以互相验证。

内部的可复制集验证可以通过2个机制来实现：

- 密钥文件验证；
- X509验证。

两种情况下，每个可复制集节点作为内部用户验证其他节点，有足够的权限来让复制工作正常进行。

密钥文件验证

更简单并且更弱安全的验证机制就是密钥文件验证。这里需要为每个节点创建一个密钥文件。密钥文件包含可复制集节点，用来验证其他节点的密钥。这个方法的优点就是易于设置，但是坏处是如果黑客攻陷了一台机器，就必须修改机器中每个节点的密码，而且完成修改工作没有停机修复时间。

开始，要创建包含密钥的文件。文件的内容会作为密码，每个可复制集成员会使用这个密码来验证其他节点。例如，我们可以创建一个名为secret.txt的文件，并且存入一下内容（实际中不要使用以下密钥）：

```
tOps3cr3tpa55word
```

在每个可复制集机器上放置密钥文件，并且调整权限，这样它就只能被所有者访问：

```
sudo chmod 600 /home/mongodb/secret.txt
```

最后，通过指定密钥文件位置的 `--keyFile` 参数来启动可复制集成员：

```
mongod --keyFile /home/mongodb/secret.txt
```

现在已经启用了验证设置。我们想事先创建一个管理员账号，和前一节中的一样。

X509 验证

X509整数验证已经内置到OpenSSL里了，MongoDB会使用它来加密网络数据。正如我们之前提到的，获取签名整数超出了本书的范围。但是一旦创建完成，就可以用如下方式来启动每个服务器节点了：

```
mongod --replSet myReplSet --sslMode requireSSL --clusterAuthMode x509 --  
sslClusterFile --sslPEMKeyFile server.pem --sslCAFile ca.pem
```

server.pem是由ca.pem对应的证书颁发机构签名的证书。

也可以不关机把密钥文件验证升级到X509验证方式。可以参考MongoDB的官方文档来查询如何操作，或者使用最新的MMS文档来看看是否已经添加到MMS自动化功能中了。

13.4.5 分片集群验证

分片集群验证是可复制集群验证的进一步扩展。分片集群中的每个可复制集使用前一节介绍的方式保护起来。此外，所有的配置服务器和mongos也可以使用与其他机器相同的验证方式，如使用密钥文件或者X509整数验证。一旦完整设置，整个集群就可以使用验证了。

13.4.6 企业安全特性

一些安全特性只在MongoDB付费的企业版插件里支持。例如，允许MongoDB与Kerberos和LDAP的验证和授权机制。此外，企业模块添加了审计支持，这样与安全相关的事件可以被追踪和记录日志。MongoDB明确介绍了这些企业版的特性。

13.5 管理任务

Administrative tasks

本节里，我们将会介绍一些基本的管理任务，包含导入和导出数据、处理磁盘碎片以及升级系统。

13.5.1 数据导入和导出

如果把现有系统迁移到MongoDB，或者需要从数据仓库里导入数据到数据库里，我们就需要一个高效的导入方法。也许需要一个好的导出策略，因为我们可能需要从MongoDB里导出数据给外部的系统。例如，导出数据给Hadoop进行批处理。这是一种常见的操作^[1]。

^[1]也有一个MongoDB的Hadoop插件，称为MongoDB Connector for Hadoop。

这里是MongoDB导出和导出数据的两种方式：

- 使用工具mongoimport和mongoexport。

- 使用驱动编写简单的程序^[1]。

mongoimport 和 mongoexport

MongoDB 提供了两个导出和导入工具：mongoimport 和 mongoexport。可以使用 mongoimport 导入JSON、CSV、TSV文件。这对于从关系型数据库导入数据到MongoDB非常有用：

```
$ mongoimport -d stocks -c values --type csv --headerline stocks.csv
```

在这个例子里，我们导入了一个名字叫stocks.csv的CSV到stocks数据库的values集合里。--headerline 标志表示CSV的第一行包含字段名字。我们可以通过运行 mongoimport--help 查看所有的导入参数。

使用mongoexport工具把集合的所有数据导出到JSON 或者CSV文件里：

```
$ mongoexport -d stocks -c values -o stocks.csv
```

这个命令导出数据到stocks.csv文件里。与其对应，我们可以通过运行mongoexport --help 来查询所有的参数。

自定义导入和导出脚本

当数据相对整齐时，我们很可能使用MongoDB的导入和导出工具。一旦引入了子文档和数组，CSV格式变得复杂，因为它并非是设计用来表示嵌套数据的。

当需要导出数据到CSV文件或者导入一个CSV文件到MongoDB中时，更简单的方式可能是使用自定义工具。我们可以使用任意的驱动来开发。例如，MongoDB用户通常都会编写脚本来连接关系型数据库，然后组合两个表的数据到一个集合中。

这也是MongoDB迁移数据时让人头疼的地方：因为不同系统之间的数据建模方式不同。这些情况下，准备使用驱动来开发自定义工具。

13.5.2 压缩和修复

MongoDB包含修复数据库的内置工具。我们可以在命令行里初始化它来修复数据库：

```
$ mongod --repair
```

或者我们也可以运行repairDatabase命令来修复单个数据库：

^[1]也可以使用 mongoconnector 工具来保持不同存储系统与 MongoDB 同步数据。


```
> use cloud-docs
> db.runCommand({repairDatabase: 1})
```

修复过程是离线进程的。当数据库运行时，会被读/写操作锁住。修复过程通过读/写数据文件来实现，抛弃过程中的冲突文档。它也会重建索引。这意味着修复数据库需要足够的磁盘空间来存储重写的数据。所以修复过程是非常昂贵的，修复大的数据库可能耗费几天时间，并且影响同节点服务器上其他数据库的请求处理工作。

MongoDB的修复最初是作为修复数据库冲突的最后一线希望。在不干净的关闭操作后，没有启用日志，修复是还原数据文件到一致状态的方式，虽然可能丢失数据。

幸运的是，如果使用复制来部署MongoDB，至少有一个服务器启用了日志，并且自定常规的备份，应该不需要修复。依赖于修复的恢复是非常傻的，所以要尽量避免。

那么修复数据库的好处是什么呢？运行修复可能会压缩数据文件并重建索引。MongoDB2.0还没有很棒的压缩文件的机制。如果执行太多的随机删除，特别是删除小文档（小于4 KB的），则尽管删除了数据，总的存储大小可能不变或者反而增加。压缩数据文件是删除碎片空间的有效方式^[1]。

如果没有时间和资源来进行完全的修复，则还有两个选择，两个都操作在单个集合上：

- 重建索引；
- 压缩集合。

要重建索引，使用`reIndex()`方法：

```
> use cloud-docs
> db.spreadsheets.reIndex()
```

这种方法可能有用，但是通常来说，索引空间是可以高效地重用的。数据文件的空间可能是个问题，所以`compact`命令通常是个更好的选择。

使用`compact`命令会重写数据文件，并且重建集合的所有索引。以下是如何在shell里运行它的例子：

```
> db.runCommand({ compact: "spreadsheets" })
```

这个命令可以运行在活动的从节点上，这样就避免了停机。一旦对所有从节点全部进行压缩，就可以继续压缩主节点。如果必须在主节点上运行此命令，就通过添加`{force: true}`到命令对象里来执行。注意，如果这样操作就会锁住系统：

```
> db.runCommand({ compact: "spreadsheets", force: true })
```

在WiredTiger数据中，执行`compact()`命令会释放不需要的磁盘空间给操作系统。也要注意`paddingFactor`字段，它适用于MMAPv1存储引擎，在使用WiredTiger存储引擎的时候没有

^[1] 压缩磁盘数据可能会导致更高效地使用内存空间。

作用。

13.5.3 升级

对于任意的软件项目，我们应该保证MongoDB部署最新的版本，因为新版本会包含重要的bug修复，还有功能改进。

MongoDB背后的一个重要设计原则就是确保升级时无需关机。对于可复制集，这意味着滚动升级；对于分片集群，这意味着mongos路由仍然可以在混合集群中正常工作。

我们可以选择两种方式之一进行升级：

- 第一个选择是手动完成。此时应该检查最新的发布提示，并自习阅读升级过程。有时候要有一些重要的工作或者步骤来确保升级安全。它的优势是我们可以完全控制升级过程，并且知道什么时候发生什么问题。
- 第二个选择就是使用MongoDB的管理服务（MMS）。MMS自动化工具不仅可以用来配置节点，还可以用来升级节点。确保在升级之前仔细阅读了发布说明和MMS文档，确保已经完全了解了升级可能带来的问题。

13.6 性能故障排除

Performance troubleshooting

MongoDB的性能问题十分复杂，它几乎涉及每个方面。在专门的负载和环境测试之前几乎不可能知道程序如何精确地执行。Joyent Cloud与EC2的服务器节点的性能完全不同，它们与私有云服务器有完全不同的性能特征。

在本节里，我们将会处理一些MongoDB支持中常见的问题，至少可以到其3.0版本。这些都是值得深思的问题。首先，我们会重新介绍工作集的概念，然后会介绍工作集影响系统的2个方式：性能悬崖和查询交互。

最后，就是监控系统，一定要理解如何监控。良好的监控可以快速解决甚至预测并且阻止性能问题，不至于等到愤怒的用户告诉你系统出现了问题再来修复。

13.6.1 工作集

本书里很多地方介绍了工作集的概念，但是我们在这里会重新定义一次，主要关注在生产环境的部署上。

假想我们有一台8GB内存的机器，使用16GB空间运行数据库，不包含索引。工作集是指在指定的时间间隔里访问多少数据。在这个例子里，如果查询都是全集合的扫描，工作集就是

16 GB，因为要响应这些查询，整个数据库都必须加载进入内存。

但是，如果查询已经建立了恰当的索引，并且，绝大部分数据可以留在磁盘上，那么内存只需要2GB的空间，在加上一些额外的索引空间。

图13.4 所示为直观显示的磁盘使用情况。

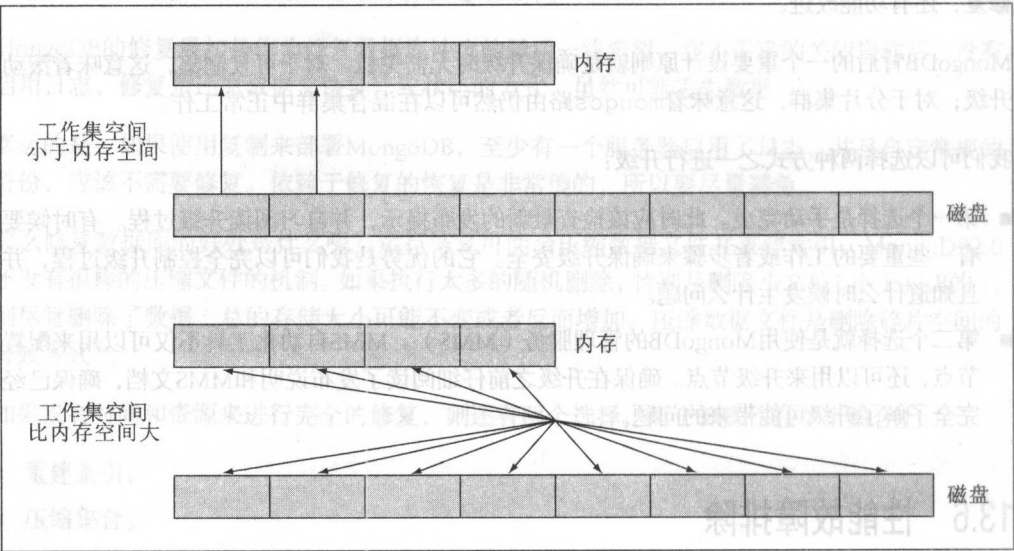


图 13.4 当工作集空间超过内存空间时对于磁盘使用的影响

在图示的下面的例子里，当我们做全表扫描时，会出现很多颠簸，或者数据块移入、移出内存，因为我们无法把16GB的数据全部一次性加载到8GB内存中。

在上面的例子中，我们可以加载2GB的数据到内存中，处理查询请求，最小化磁盘访问。这个不仅演示了为什么把工作集保存到内存多么的重要，还展示了一些简单的变化，比如添加正确的索引，可以完全改变性能特性。

13.6.2 性能悬崖

与工作集概念紧密相关的就是MongoDB运行查询的方式。MongoDB3.0没有限制单个操作使用的资源^[1]，而且在超负荷时也不会显示退回给客户端。只要没有达到MongoDB的限制，这就不是个问题。但是到达限制时，可能会看到“性能悬崖”。

要弄清楚为什么会这样，就想像一下，系统有8GB内存，运行的数据库在磁盘上有64GB数据，工作集只有6GB，并且应用执行得非常好。现在，流量高峰足够达到8GB。MongoDB将会开

^[1]一个粗略的近似值是 maxTimeMS 光标参数，它可以设置处理操作的最大时间。不会阻止资源饥饿操作的交互，但是它会杀死运行时间超过期望的操作。

始变慢，因为工作集无法完全加载进入内存，并且会频繁在磁盘上导入/导出数据。除了这些问题，MongoDB还要接受新的请求。

系统变慢再加上继续接受新请求，这对于性能来说无异于雪上加霜，我们可能会看到性能明显下滑。如果关注这个问题，大家就必须精确掌握自己部署的MongoDB服务器可以处理的最大压力。如果可能，可以借助负载均衡器来分担压力，这样可以避免系统达到性能瓶颈，出现性能悬崖问题。

13.6.3 查询交互

MongoDB不限制查询资源使用的潜在问题就是，一旦出现恶性查询就可能影响系统上所有的其他查询。

还是之前的练习。假设工作集是2GB，有64GB的数据库。运行一切正常，直到有个人执行了全集合查询。这个查询不仅会在磁盘上施加大量的压力，还会输出结果给系统其他查询使用，当然肯定会引起系统性能变慢。图13.4演示了这个问题，上面表示正常的查询压力，下面表示糟糕查询之后的压力。

这也是为什么访问控制非常重要的另外一个原因。即使其他设置都正常，一个表扫描就可以拖慢整个系统。确保每个有查询数据库权限的人明白糟糕查询的后果^[1]。

关于索引

当发现性能问题时，索引应该是第一求助方案。除非操作是插入，索引是保证良好性能的关键所在。

第8章介绍了查找缓慢查询以及启用查询分析器来修复问题的过程，确保每个查询都可以高效地使用索引。通常，这意味着每个操作都会扫描尽可能少的文档。

还有一点非常重要，要确保没有冗余的索引，因为冗余索引会占据磁盘空间，需要更多的内存，并且每个写入需要更多的资源。第8章介绍了减少索引冗余的方法。

然后呢？在审查了索引和查询以后，你可能会发现性能问题已经解决了。日志里再也看不到缓慢查询的警告信息，iostat输出结果显示了更少的资源使用率。

调整索引修复性能问题是经常使用的方法，比我们想象的要多。在处理性能问题时，这应该是首选策略。

^[1]我们可以把查询压力分散到从服务器上。但是记住，第11章里介绍了从服务器必须有足够的带宽可以与主服务器保持同步，否则会落后同步操作日志 oplog。

13.6.4 寻求专业帮助

性能下降的原因多种多样，而且经常发生。糟糕的数据定义和服务器bug都会影响性能。

如果你认为已经尝试了许多问题，仍然无法修复，就要考虑求助有经验的专家来帮助审查MongoDB所在的系统。书本只提供理论参考，而经验要通过实践检验。当我们无所适从、心存疑问时，就可以寻求专业的帮助。这个性能问题的解决方案有时候完全不够直观。

当寻求帮助或需要帮助时，记得提供问题发生的所有信息。这就是监控日志的价值。MongoDB官方使用的是MMS监控工具，所以如果你也在使用MongoDB支持，使用MMS监控工具可以显著加快问题的解决过程。

13.7 部署检查列表

Deployment checklist

本章我们已经介绍了许多主题。开始一看可能有点晕头转向。其实只要掌握了主要的知识点，就能让系统平稳运行。本节里我们会快速看一下如何确保大家已经掌握了重要的知识点。

■ 硬件：

内存——足够处理期望的工作集。

磁盘空间——足够的空间去处理所有的数据、索引和MongoDB内部元数据。

磁盘速度——足够满足延迟和吞吐量的需求。与内存一起考虑：更少的内存意味着更多的磁盘空间。

CPU——通常不是MongoDB的瓶颈，但如果低磁盘使用率、低吞吐量，就可能是CPU负荷的问题。可以通过性能测试来检查这个问题。

网络——确保网络足够快和可靠，可以满足性能需求。MongoDB节点需要内部彼此通信，所以确保测试每个连接，不仅仅是从自己的客户端到mongos或者mongod服务器。

■ 安全：

保护网络通信——在完全隔离的环境里运行或者确保使用MongoDB内置的SSL连接支持功能，以便保护通信。

访问控制——确保只有信任的用户和客户端程序可以操作数据库。确保测试账号不要有“root”权限。

■ 监控：

硬件使用率（磁盘、CPU、网络、内存）——确保对使用的硬件资源有监控措施，不仅要监控，还要有超额提醒功能。

健康检查——进行定期检查，确保服务器运行并可以响应请求，如果有人停止了回调就通知我们。

监控——使用MMS监控服务。它不仅提供了监控、健康检查和提醒等功能，也是MongoDB支持团队用来解决问题的工具。一直以来都是免费工具，所以不要犹豫，生产环境下可以直接使用。

客户端性能监控——定期运行端到端的测试，确保客户端可以第一个知道哪个应用慢。

■ 灾难恢复：

评估风险——想象一下我们已经丢失了所有数据，该有多伤心！在所有严重的情况中，丢失数据可能是最严重的。

如果你分析Twitter的趋势时丢失数据，可能需要花费一周的时间来恢复；但是如果丢失的是银行数据，那么可能花费更多时间才能恢复。当做这种类型评估时，就已经发生了这种类型的灾难，然后据此制订计划。

制订计划——为灾难情况下的恢复制订详细的计划。根据系统失败的原因采用不同的恢复方式。

测试计划——确保测试计划。对于备份和灾难恢复，大家犯的最大错误就是以为已经有了备份或者足够的计划。这是不够的。可能备份会冲突，可能它的格式不可以重新导入生产环境下的系统中。对于生产系统，许多事情可能出错，所以确保恢复策略正常工作非常重要。

制订备份计划——第一个灾难恢复计划可能失败。真的失败时，还有最后一个方法。这不是必须的选择，但是如果有备份，就会更开心。

■ 性能：

负载测试——确保使用真实的工作负载来测试我们的应用。这是唯一确保性能能满足需求的方式。

13.8 总结

Summary

本章介绍了在生产环境里部署MongoDB的最重要知识点。大家应该掌握了正确选择硬件、监控部署、维护常规备份的所有知识。此外，我们还应该知道如何解决性能问题。最终，这些知识都需要与经验相结合。MongoDB已经足够可预测地适应这种简单的启发模式。MongoDB尝试化繁为简，但是数据库及其与应用系统的交互是十分复杂的。本章的指南可以给大家指引正确的方向，但是最后能达成怎样的效果，取决于我们如何理解自己的系统。确保已经使用了从MongoDB文档到官方社区MongoDB用户的支持等所有可用的资源。

安装

Installation

在本附录内，我们会介绍如何在Linux、Mac OS X、Windows系统安装MongoDB，并且会介绍一下MongoDB最常用的配置参数。对于开发者，从源码编译安装有一些提示要注意。

这里我们还会安装Ruby 和RubyGems，便于运行本书里的例子^[1]。

A.1 安装

Installation

在我们继续介绍安装过程之前，关于MongoDB版本的一个提示是要按照顺序。简单来说，应该为了自己的架构运行一个最新的稳定的版本。MongoDB稳定版本都标记有偶数。

因此，1.8、2.0、2.2、2.4、2.6和3.0版本是稳定的；2.1、2.3、2.5是开发版本，不应该用在生产环境下。www.mongodb.org的下载页面提供了32位和64位编译版本的静态下载链接。但是，MongoDB 3.0对于32位的二进制版本不再提供支持了。这些二进制文件都有最新的稳定发布版本，还有开发者分支的最新编译版本。二进制版本提供了MongoDB跨平台安装的简单方式，包括Linux、Mac OS X、Windows、Solaris，这也是我们推荐的方式。如果遇到问题，就可以在<http://docs.mongodb.org/manual/installation> MongoDB官方手册里查找资料。

A.1.1 生产部署

对于MongoDB服务器的部署，推荐使用包管理工具，因为包安装通常包含启动和停止MongoDB的脚本，当机器重新启动时会自动启动。在这里描述预编译的二进制文件安装，将会演示如何通过命令行运行MongoDB，这对于学习和调试来说非常完美，但是不适合作为服务器供他人访问。每个操作系统的分发和包管理系统存在差别，比如MongoDB日志文件的存放位置。如果当作服务器运行MongoDB，就应该了解MongoDB的配置并思考其错误场景，比如服务器突然关闭，会发生什么？如果是在学习MongoDB或者实验操作，则任意安装方法都可以采用。

[1]【译者注】关于Java、.NET、Node.js、.NET Core的MongoDB例子，可以加入《MongoDB 实战》中国读者交流群 511943641。

A.1.2 32 位和 64 位

我们可以根据MongoDB版本来下载32位或者64位架构的编译版本。如果你正在64位机器上安装MongoDB,就强烈推荐使用64位版本,因为32位安装版本只能存储2 GB的数据。现在绝大多数机器都是64位的,但是如果不确定Linux 或者Mac OS X是什么版本,则可以运行下面的命令来查看:

```
$ uname -a
```

输出的是系统的版本信息。如果OS版本包含x86_64 (与i386相反),然后运行64位版本。对于Windows系统,直接检查我的电脑=>属性,就可以看到系统的版本。

A.2 Linux 下安装 MongoDB

MongoDB on Linux

Linux下安装MongoDB有三种方式。我们可以直接在mongodb.org官方网站下载预编译的二进制文件安装,或者使用包管理器安装,或者手动编译源代码安装。我们会在下一节里讨论前两种方式,然后在后面接受编译安装方式。

A.2.1 使用预编译二进制文件安装

打开www.mongodb.org/downloads网页,就会看到所有可用的MongoDB二进制文件。选择最新的稳定版本再点击下载。这里的例子使用的是MongoDB 2.6编译的64位版本^[1]。

打开命令行,使用浏览器下载文件,curl或者wget工具都可以(最好检查下载页面,确定最新的版本)。然后使用tar解压文件:

```
$ curl http://downloads.mongodb.org/linux/mongodb-linux-x86_64-2.6.7.tgz >
mongo.tgz
$ tar xzvf mongo.tgz
```

要运行MongoDB,需要一个数据目录。默认情况下,mongod daemon会在/data/db目录下存储数据。创建目录文件夹,并且确保它有对应的权限:

```
$ sudo mkdir -p /data/db/
$ sudo chown `id -u` /data/db
```

现在可以准备启动服务器了。修改MongoDB bin目录,然后启动mongod.exe:

```
cd mongodb-linux-x86_64-2.6.7/bin
./mongod
```

如果一切正常,应该可以看到下面的启动日志信息。第一次启动服务器可能要分配日志文件,

[1]【译者注】我们建议使用最新的 MongoDB 3.2.x 版本。大家安装时尽量选择最新的稳定版本。

因此在准备接受连接之前会花费几分钟。注意最后一行，确认服务器侦听在默认的27017端口上：

```
Thu Mar 10 11:28:51 [initandlisten] MongoDB starting :
pid=1773 port=27017 dbpath=/data/db/ 64-bit host=iron
Thu Mar 10 11:28:51 [initandlisten] db version v2.6.7
:
Thu Mar 10 11:28:51 [websvr] web admin console waiting for connections on
port 28017
Thu Mar 10 11:28:51 [initandlisten] waiting for connections on port 27017
```

现在我们应该可以运行./mongo使用Javascript控制台来连接MongoDB服务器了。如果服务器意外中断，参考附录A.6。

在编写本书的时候，最近的MongoDB版本号是3.0.6。

A.2.2 使用包管理器

使用包管理器可以大大简化MongoDB的安装工作。唯一的缺点是包管理器软件下载的都是最新的MongoDB版本。非常重要是运行最新的稳定的版本，所以如果选择使用包管理器，就要确认安装的是最新版本。

如果在Debian、Ubuntu、CentOS、Fedora等系统上安装，通常要访问最新的版本。这是因为MongoDB公司维护并发布这些平台专用的包。我们可以在mongodb.org官网找到更多信息。Debian和Ubuntu的安装指南可以在<http://mng.bz/ZffG>找到。CentOS 和 Fedora可以参考<http://mng.bz/JsjC>。

也可以找到FreeBSD 和 ArchLinux的安装包。可以单独查看对应的包存储库。还可能有的包管理器包含了MongoDB，但是这里没有列出来。检查一下下载页面www.mongodb.org/downloads可获取更多信息。

A.3 Mac OS X 下安装 MongoDB

MongoDB on Mac OS X

如果使用的是Mac OS X系统，则有三种方式可以安装MongoDB。可以直接在mongodb.org官网下载二进制预编译文件安装，也可以使用包管理器安装，或者手动编译源码安装。我们将在接下来讨论前面两种方式，然后提供一些编译提示。

A.3.1 预编译二进制版本

首先打开www.mongodb.org/downloads，会看到所有可以下载的MongoDB二进制包。选择最

新的稳定版本的下载链接。下面的例子使用的是64位系统的MongoDB 3.0.6。

使用浏览器或者curl工具来下载压缩文件。最好检查一下最新发布的稳定版本，然后再使用tar解压压缩包：

```
$ curl https://fastdl.mongodb.org/osx/mongodb-osx-x86_64-3.0.6.tgz >
mongo.tgz
$ tar xzvf mongo.tgz
```

要运行MongoDB，就需要一个数据目录。默认mongod daemon会在/data/db存储数据。继续使用命令创建文件夹：

```
$ sudo mkdir -p /data/db/
$ sudo chown `id -u` /data/db
```

现在都准备启动服务器了。切换路径到MongoDB bin文件夹，然后启动mongod：

```
$ cd mongodb-osx-x86_64-3.0.6/bin
$ ./mongod
```

如果一切正常，就应该可以看到下面的启动日志信息。第一次启动服务器可能要分配日志文件，因此在准备接受连接之前会花费几分钟。注意最后一行，确认服务器侦听在默认的27017端口上：

```
2015-09-19T08:51:40.214+0300 I CONTROL [initandlisten] MongoDB starting :
pid=41310 port=27017 dbpath=/data/db 64-bit host=iron.local
2015-09-19T08:51:40.214+0300 I CONTROL [initandlisten] db version v3.0.6
:
2015-09-19T08:51:40.215+0300 I INDEX [initandlisten] allocating new ns
file /data/db/local.ns, filling with zeroes...
2015-09-19T08:51:40.240+0300 I STORAGE [FileAllocator] allocating new
datafile /data/db/local.0, filling with zeroes...
2015-09-19T08:51:40.240+0300 I STORAGE [FileAllocator] creating directory /
data/db/ tmp
2015-09-19T08:51:40.317+0300 I STORAGE [FileAllocator] done allocating
datafile /data/db/local.0, size: 64MB, took 0.077 secs
2015-09-19T08:51:40.344+0300 I NETWORK [initandlisten] waiting for
connections on port 27017
```

现在我们应该可以运行./mongo使用JavaScript控制台来连接MongoDB服务器了。如果服务器意外中断，参考附录A.6。

A.3.2 使用包管理器

MacPorts (<http://macports.org>) 和 Homebrew (<http://brew.sh/>) 是Mac OS X系统上的2个包管理器，维护了最新的MongoDB版本。要安装MacPorts，运行以下命令即可：

```
sudo port install mongodb
```

注意：MacPorts会编译MongoDB和它所有的依赖包。如果用这种方式，需要很长的编译时间。

Homebrew不同，不需要编译，只需要下载最新的二进制包，所以它比MacPorts更快。我们可以通过Homebrew直接安装MongoDB，命令如下：

```
$ brew update
$ brew install mongodb
```

完成安装以后，Homebrew会提供如何在Mac OS X上启动MongoDB的指令。

A.4 Windows 下安装 MongoDB

MongoDB on Windows

如果使用的是Windows系统，有两种方式可以安装MongoDB。简单、推荐的方式是直接从mongodb.org下载预编译的二进制文件进行安装。也可以下载源码自己编译，但是这个方式只推荐开发者和高级用户使用。在接下来一节里会详细介绍编译的过程。

首先打开www.mongodb.org/downloads，会看到所有可以下载的MongoDB二进制包。选择最新的稳定版本的下载链接。下面例子使用的是MongoDB 2.6编译的64位版本。

下载正确的分发版，然后解压。可以直接在MongoDB资源管理器里打开MongoDB压缩文件，右击进行解压后选择解压后的文件。记住，MongoDB 2.6版本中通过预构建的MSI也可以下载，直接点击即可安装。

另外也可以使用命令行下载。首先切换到下载目录，然后使用unzip命令解压文件：

```
C:\> cd \Users\kyle\Downloads
C:\> unzip mongodb-win32-x86_64-2.6.7.zip
```

要运行MongoDB，需要一个数据文件夹。默认mongod daemon在C:\data\db下创建文件。打开命令行窗口，使用命令来创建文件夹：

```
C:\> mkdir \data
C:\> mkdir \data\db
```

现在可以准备启动服务器了。切换路径到MongoDB bin目录，然后启动mongod.exe：

```
C:\> cd \Users\kyle\Downloads
C:\Users\kyle\Downloads> cd mongodb-win32-x86_64-2.6.7\bin
C:\Users\kyle\Downloads\mongodb-win32-x86_64-2.6.7\bin> mongod.exe
```

如果一切正常，应该可以看到下面的启动日志信息。第一次启动服务器可能要分配日志文件，这样会花费几分钟。注意最后一行，确认服务器侦听在默认的27017端口上：

```
Thu Mar 10 11:28:51 [initandlisten] MongoDB starting :
pid=1773 port=27017 dbpath=/data/db/ 64-bit host=iron
Thu Mar 10 11:28:51 [initandlisten] db version v2.6.7
:
Thu Mar 10 11:28:51 [websvr] web admin console waiting for connections on
```

```
port 28017
```

```
Thu Mar 10 11:28:51 [initandlisten] waiting for connections on port 27017
```

如果服务器意外中断，就参考附录A.6。

首先要启动MongoDB shell客户端。为此，可以打开终端窗口，然后输入mongo.exe启动：

```
C:\> cd \Users\kyle\Downloads\mongodb-win32-x86_64-2.6.7\bin
```

```
C:\Users\kyle\Downloads\mongodb-win32-x86_64-2.6.7\bin> mongo.exe
```

A.5 从源码编译 MongoDB

Compiling MongoDB from source

从源码编译MongoDB的方式只推荐给开发者和高级用户。如果你喜欢最新的版本而不需要编译源码，则可以直接从mongodb.org官网下载最新的二进制文件。

当然也可以自己编译。编译MongoDB最复杂的部分是管理各种不同的依赖包。各种不同平台的最新的编译指令可以在www.mongodb.org/about/contributors/tutorial/build-mongodb-from-source找到。

A.6 故障排除

Troubleshooting

MongoDB安装非常简单，但是用户经常遇到一些问题。这些问题通常是在启动mongod时出现错误。这里我们提供了一个常见错误和解决方法的列表。

A.6.1 错误的架构

如果在32位系统上运行64位的安装包，就会看到下面的出错提示信息：

```
bash: ./mongod: cannot execute binary file
```

在Windows 7系统上，出错提示信息比较有帮助：

This version of

```
C:\Users\kyle\Downloads\mongodb-win32-x86_64-2.6.7\bin\mongod.exe
```

is not compatible with the version of Windows you're running.

Check your computer's system information to see whether you need a x86 (32-bit) or x64 (64-bit) version of the program, and then contact the software publisher.

针对这两种情况的解决办法是一样的，换成32位版本即可。在MongoDB官网可以下载 (www.mongodb.org/downloads)。

A.6.2 不存在的数据目录

MongoDB需要一个文件夹存储数据文件。如果文件夹不存在，就会报错：

```
dbpath (/data/db/) does not exist, terminating
```

解决办法是创建这个目录。具体操作可以查询自己操作系统的指南。

A.6.3 缺少权限

如果在UNIX上运行，则要确定运行mongod时有足够的写数据目录的权限。否则就会看到这个出错提示：

```
Permission denied: "/data/db/mongod.lock", terminating
```

或者是这个

```
Unable to acquire lock for lockfilepath: /data/db/mongod.lock, terminating
```

以上每一种情况都可以通过chmod或chown打开数据目录的权限来实现。

A.6.4 未绑定端口

MongoDB默认运行在27017端口上。如果另外一个进程或者mongod正在使用同一个端口，就可以看到这个出错提示：

```
listen(): bind() failed errno:98  
Address already in use for socket: 0.0.0.0:27017
```

这个问题有2个解决办法。第一个是找到使用此端口的进程，然后停止它。查找侦听在端口27017的进程的方法如下：

```
sudo lsof -i :27017
```

Lsof工具还会返回侦听端口27017进程的ID信息，这个可以用kill来终止进程。

另外一种方式是使用-port标志来使用不同的端口启动mongod，这是个简单、方便的解决办法。以下是在27018端口运行MongoDB的命令：

```
mongod --port 27018
```

A.7 基本配置选项

Basic configuration options

这里我们来一起看看运行MongoDB时最常见的参数选项：

- `--dbpath`——数据文件存储的目录。默认是/data/db，如果要设置其他MongoDB数据文件目录，就可以使用这个参数。
 - `--logpath`——这个路径是指定日志文件存储的目录。默认日志输出打印信息在标准输出窗口(stdout)。
 - `--port`——指定MongoDB 侦听的端口，默认是27017。
 - `--rest`——此标志用来启用Web控制台的简单REST接口。Web控制台默认比MongoDB 侦听的端口大1000。例如，如果MongoDB 侦听在27017端口，Web控制台可以在http://localhost:28017地址。
- 花点时间研究Web控制台和它提供的命令，就可以发现更多关于运行状态的MongoDB服务器的信息。
- `--fork`——作为守护进程分离进程。注意，fork只在UNIX分支平台使用。Windows用户如果查询相似的功能，则应该看看如何把MongoDB作为Windows服务器运行的指令。可以在www.mongodb.org官方网站找到。

这些是最重要的MongoDB启动参数标志。下面是命令行启动方式的例子：

```
$ mongod --dbpath /var/local/mongodb --logpath /var/log/mongodb.log
--port 27018 --rest -fork
```

注意：也可以在配置文件里指定这些参数。创建一个文本文件(叫做 mongodb.conf)，可以在配置文件里指定所有等价的参数^[1]：

```
storage:
  dbPath: "/var/local/mongodb"
systemLog:
  destination: file
  path: "/var/log/mongodb.log"
net:
  port: 27018
  http:
    RESTInterfaceEnabled: true
processManagement:
  fork: true
```

可以使用-f参数附加配置文件来启动mongod：

```
$ mongod -f mongodb.conf
```

如果已经连接了MongoDB，并且想知道哪些参数可以用来启动数据库，则可以通过getCmdLineOpts命令来查询：

```
> use admin
> db.runCommand({getCmdLineOpts: 1})
```

^[1]对于 MongoDB，使用 YAML 配置文件格式。参考 docs.mongodb.org/manual/reference/configurationoptions 获取更多可用的参数选项。

A.8 安装 Ruby

Installing Ruby

本书中很多例子是使用Ruby编写的，所以要运行这些代码，就要安装Ruby。这意味着需要安装Ruby解释器和Ruby包管理器RubyGems。^[1]

A.8.1 Linux 和 Mac OS X

Mac OS X和许多Linux分支默认安装了Ruby。也可以通过运行下面的命令来检查是否是最新的版本：

```
ruby -v
```

如果没有找到这个命令，或者运行的版本低于1.8.7，则表示可能要升级了。Mac OS X和UNIX分支的安装文档可以在<https://www.ruby-lang.org/en/downloads/>(也可以下拉看看其他平台的安装指南)找到。

绝大多数包管理器（比如MacPorts 和Aptitude）也维护了最新的Ruby版本，通过这些包管理器安装Ruby也十分方便。

为了安装MongoDB Ruby驱动，除了Ruby解释器外，还需要安装Ruby包管理器RubyGems。通过运行gem命令来确定是否已经安装RubyGems：

```
gem -v
```

也可以通过包管理器安装RubyGems，但是绝大多数用户会下载最新版本并使用其中的安装器安装。我们也可以在<https://rubygems.org/pages/download>下载。

A.8.2 Windows

到目前为止，在Windows上安装Ruby和RubyGems的最简单的方式是使用Windows Ruby安装器。可以在这里下载：<http://rubyinstaller.org/downloads>。

应该可以安装新版本Ruby，比如1.9.3或者2.2.3，这是当前的稳定版本。许多人还在使用1.8.7版本，这当然也可以正常使用MongoDB，但是新版本的Ruby有许多优点，比如更好的字符编码，值得升级。

当运行可执行文件时，向导会指引我们完成Ruby和RubyGems的安装过程。

除了Ruby，也可以安装Ruby DevKit开发工具包，它支持Ruby C扩展的简单编译功能。MongoDB Ruby驱动的BSON库也可以使用这个扩展选择安装。

^[1]【译者注】Java、C#、Node.js等语言例子，加《MongoDB 实战》中国交流群 511943641 索取。

设计模式

Design patterns

本书的前面几章含蓄提出了一些设计模式。

这里我们会总结一下这些设计模式，并且会介绍一些其他的设计模式。

B.1 嵌入与引用

Embed vs. reference

假设我们在使用MongoDB构建一个简单的网站，包含博客和评论。那么如何表示这些数据？要不要在每个博客里单独嵌入它的评论？或者创建两个集合，一个是博客一个是评论，然后在两者之间使用ID(_id)进行引用关联？

这是嵌入式和引用的对比问题，而且这是MongoDB新用户常见的困惑。幸运的是，对于这种最常见的设计问题有个简单的规则：当子对象从不会出现在父辈以外的环境中时使用嵌入方式。否则在单独的集合里存储子对象。

如果评论一直出现在博客文章里，并且不会按照某种方式排序（发布日期、评论分级等），则使用嵌入方式比较好。但是如果要显示最近的评论，不关注它们在每个评论里，则可以使用引用方式。嵌入提供了一些性能优势，而引用提供了更多的灵活性。

B.2 一对多

One-to-many

正如上一节介绍的，我们可以使用嵌入或者引用来表示一对多关系。当许多对象只属于父对象并且很少修改时可以使用嵌入方式。这种数据定义的典型代表就是How-To网站，是介绍如何操作的文档类网站。每个指南中的步骤可以使用子文档数组来表示，因为这些步骤都是指南的一部分而且很少改变：

```
{ title: "How to soft-boil an egg",
  steps: [
    { desc: "Bring a pot of water to boil.",
      materials: ["water", "eggs"] },
```

```
{ desc: "Gently add the eggs a cook for four minutes.",
  materials: ["egg timer"]},
{ desc: "Cool the eggs under running water." },
}
```

注意，如果要确保数组的顺序，就可能要添加额外的字段来保存元素的顺序，因为某些语言不保证数组的顺序。

当两个相关的实体独立出现在应用中时，我们想要关联它们。许多关于MongoDB的文章建议采用嵌入评论的方式，这是个不错的主意。但是关联方式更加灵活。因为可以方便地向用户展示它们的评论列表——使用稀疏索引可能有帮助，因为稀疏索引值包含索引字段文档的入口。我们也可以显示所有文章的所有评论。这些功能是网站的常见功能，此时使用嵌入式文档可能无法实现^[1]。

可以使用对象ID来关联文档。以下是个简单的例子：

```
{ _id: ObjectId("4d650d4cf32639266022018d"),
  title: "Cultivating herbs",
  text: "Herbs require occasional watering..."
}
```

这里是评论，通过post_id字段关联：

```
{ _id: ObjectId("4d650d4cf32639266022ac01"),
  post_id: ObjectId("4d650d4cf32639266022018d"),
  username: "zjones",
  text: "Indeed, basil is a hearty herb!"
}
```

文章和评论在单独的集合里，而且需要两个查询来显示文章和评论。因为要根据post_id查询文章的所有评论，这里需要一个索引：

```
db.comments.createIndex({post_id: 1})
```

这里我们使用了第4、5、6章里介绍的一对多模式；可以在这里查看更多的例子。

B.3 多对多

Many-to-many

在RDBMS中，我们使用连接表来表示多对多管理；在MongoDB里，我们使用数组键来表示。本书的前面有个例子就使用了这种方式来关联商品和类别。每个商品包含一个类别ID的数组，而且商品和类别有专门的集合。如果有两个简单的类别文档：

```
{ _id: ObjectId("4d6574baa6b804ea563c132a"),
```

^[1] 还有一个关于虚拟集合(virtual collections)的功能请求，问题追踪请参考网页 <http://jira.mongodb.org/browse/SERVER-142>。

```
title: "Epiphytes"
}
{ _id: ObjectId("4d6574baa6b804ea563c459d"),
title: "Greenhouse flowers"
}
```

商品属于两个类别，表示的文档如下所示：

```
{ _id: ObjectId("4d6574baa6b804ea563ca982"),
name: "Dragon Orchid",
category_ids: [ ObjectId("4d6574baa6b804ea563c132a"),
ObjectId("4d6574baa6b804ea563c459d") ]
}
```

则为了高效查询，可以在类别ID上建立索引：

```
db.products.createIndex({category_ids: 1})
```

然后查询所有Epiphytes类别下的商品，根据category_id字段匹配：

```
db.products.find({category_id: ObjectId("4d6574baa6b804ea563c132a")})
```

现在返回所有Dragon Orchid商品的类别文档信息，首先来获取商品的类别ID：

```
product = db.products.findOne({_id: ObjectId("4d6574baa6b804ea563c132a")})
```

然后使用\$in操作符查询categories集合：

```
db.categories.find({_id: {$in: product['category_ids']}})
```

我们会发现查找类别需要2次查询，商品搜索需要1次查询。它为常见的情况做了优化，我们更愿意在某个类别里搜索商品，这是比较常见的方式。

B.4 树

Trees

与MySQL类似，MongoDB也没有内置工具来表示和遍历树形数据结构——Oracle 有 CONNECT BY、PostgreSQL 有 WITH RECURSIVE来遍历树。其解决办法是在每个类别里存储一个父类ID。这种反范式做法使得更新复杂化，但是大大简化了读操作。

这种做法并非万能的。另外一个常见的树形结构是在线论坛，几百条消息频繁地内嵌到不同级别。使用祖先方法需要太多内嵌数据，好的替代做法是物化路径。

遵守物化路径模式，每个树节点都包含一个path字段，这个字段存储了每个级联节点祖先的ID，根节点path为空，因为没有祖先。我们来看一个例子了解它的工作原理。首先，我们看一下图B.1的帖子和评论。这个是关于希腊历史的问答帖子。

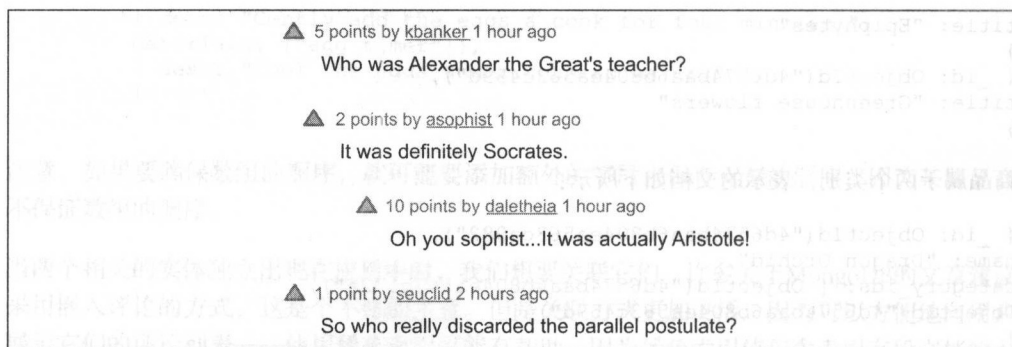


图 B.1 论坛里的帖子评论

我们来看看这些评论使用物化路径方式的文档阻止形式。

首先来看一下根节点的文档，path为空：

```
{ _id: ObjectId("4d692b5d59e212384d95001"),
  depth: 0,
  path: null,
  created: ISODate("2011-02-26T17:18:01.251Z"),
  username: "plotinus",
  body: "Who was Alexander the Great's teacher?",
  thread_id: ObjectId("4d692b5d59e212384d95223a")
}
```

其他根级别的问题、用户seuclid的文档，也有相似的结构。

更直观的例子是关于伟大老师Alexander的评论。检查这些评论，看看路径包含中间的节点_id：

```
{ _id: ObjectId("4d692b5d59e212384d951002"),
  depth: 1,
  path: "4d692b5d59e212384d95001",
  created: ISODate("2011-02-26T17:21:01.251Z"),
  username: "asophist",
  body: "It was definitely Socrates.",
  thread_id: ObjectId("4d692b5d59e212384d95223a")
}
```

接下来更深入的评论路径包含了最初和中间评论的ID，按照顺序由冒号分割：

```
{ _id: ObjectId("4d692b5d59e212384d95003"),
  depth: 2,
  path: "4d692b5d59e212384d95001:4d692b5d59e212384d951002",
  created: ISODate("2011-02-26T17:21:01.251Z"),
  username: "daletheia",
  body: "Oh you sophist...It was actually Aristotle!",
  thread_id: ObjectId("4d692b5d59e212384d95223a")
}
```

我们希望在thread_id 和path字段上建立字段，因为这是后期查询最频繁的字段：

```
db.comments.createIndex({thread_id: 1})
db.comments.createIndex({path: 1})
```

现在的问题是如何查询并显示树形结构。无论要显示全部评论还是只是子树节点，物化路径模式的一大优势就是只需查询一次。查询根节点非常简单：

```
db.comments.find({thread_id: ObjectId("4d692b5d59e212384d95223a")})
```

查询特定的子树也很简单，因为只需要使用前缀查询（第5章介绍了）：

```
db.comments.find({path: /^4d692b5d59e212384d95001/})
```

它会返回以特定字符开始的路径的所有评论。这个字符串代表用户名为plotinus的评论_id，如果查询每个子评论的path字段，就会轻易发现这些完全满足这个查询，而且这个查询非常快，因为前缀查询可以使用path上的索引机制。

获取评论列表也非常容易——只需要一次查询。显示评论比较复杂，因为需要一个保存帖子顺序的列表。这需要一些客户端处理，可以使用Ruby方法来存档^[1]。

第一个方法threaded_list构建一个根节点评论的列表，还有一个保存父节点ID列表和子节点关系的map数据结构：

```
def threaded_list(cursor, opts={})
  list = []
  child_map = {}
  start_depth = opts[:start_depth] || 0
  cursor.each do |comment|
    if comment['depth'] == start_depth
      list.push(comment)
    else
      matches = comment['path'].match(/([d|w]+)$/i)
      immediate_parent_id = matches[1]
      if immediate_parent_id
        child_map[immediate_parent_id] ||= []
        child_map[immediate_parent_id] << comment
      end
    end
  end
  assemble(list, child_map)
end
```

assemble方法使用根节点列表，还有子地图，然后按照显示顺序构建新列表：

```
def assemble(comments, map)
  list = []
  comments.each do |comment|
    list.push(comment)
    child_comments = map[comment['_id'].to_s]
    if child_comments
      list.concat(assemble(child_comments, map))
    end
  end
end
```

^[1]本书的源代码包含使用物化路径方式评论的完整例子，还有显示方法。

```

end
list
end
要打印评论，只要有迭代列表即可，适当地缩进每个评论的深度depth:

def print_threaded_list(cursor, opts={})
  threaded_list(cursor, opts).each do |item|
    indent = " " * item['depth']
    puts indent + item['body'] + " #{item['path']}"
  end
end
end

```

查询评论并打印评论十分简单：

```

cursor = @comments.find.sort("created")
print_threaded_list(cursor)

```

B.5 工作队列

Worker queues

我们可以在MongoDB中使用标准或者盖子集合实现工作队列。两种情况下，findAndModify命令都可以允许我们自动处理队列入口。

队列入口需要state和timestamp再加上其他的有效负载。状态可以编码为字符串，但是采用整数更节约空间。我们会使用0和1来表示processed 和 unprocessed。时间戳是标准的BSON日期。任意的有效数据都是简单的文本消息，也可以是其他类型：

```

{ state: 0,
  created: ISODate("2011-02-24T16:29:36.697Z"),
  message: "hello world" }

```

需要声明一个索引来帮助我们高效地获取最久的未处理项目(FIFO)。用State和created两个字段建立组合索引可以很好地满足需求：

```

db.queue.createIndex({state: 1, created: 1})

```

然后可以使用findAndModify返回下一个项目，并标记为processed（处理过）：

```

q = {state: 0}
s = {created: 1}
u = {$set: {state: 1}}
db.queue.findAndModify({query: q, sort: s, update: u})

```

如果使用标准集合，就要确保删除旧的项目。使用findAndModify的{remove: true}就可以在特定的时间删除它们。但是，一旦处理完成，某些应用程序可能想推迟到某个特定的时间再删除。

盖子集合也可以作为工作队列的基础。没有在下_id上设置索引，盖子集合有更快的插入速度，

但是这个差别可能绝大多数应用不需要。其他的好处是自动删除。但是这个功能是把双刃剑：

我们必须确保盖子集合有足够大的空间以避免未处理的项目被自动删除。因此如果使用了盖子集合，就要确保它足够大。理想的集合大小依赖于队列的写入吞吐量和平均有效载荷大小。一旦确定了盖子集合的大小，数据定义、索引和findAndModify与刚才介绍的标准集合都是一样的。

B.6 动态特性

Dynamic attributes

MongoDB的文档数据模型对于表示变化的实体非常有用。商品就是典型的例子，而且本书早期也介绍了建模这些字段的一些方式。其中一种方式就是把这些动态特性保存到子文档里。在单个products集合里，可以存储不同的商品类型。可以存储商品耳机的集合：

```
{ _id: ObjectId("4d669c225d3a52568ce07646")
  sku: "ebd-123"
  name: "Hi-Fi Earbuds",
  type: "Headphone",
  attrs: { color: "silver",
            freq_low: 20,
            freq_hi: 22000,
            weight: 0.5
          }
}
```

和商品SSD磁盘的集合：

```
{ _id: ObjectId("4d669c225d3a52568ce07646")
  sku: "ssd-456"
  name: "Mini SSD Drive",
  type: "Hard Drive",
  attrs: { interface: "SATA",
            capacity: 1.2 * 1024 * 1024 * 1024,
            rotation: 7200,
            form_factor: 2.5
          }
}
```

如果需要在这些字段上频繁查询，则可以创建稀疏索引。例如，可以优化在耳机字段上频繁的范围查询：

```
db.products.createIndex({"attrs.freq_low": 1, "attrs.freq_hi": 1},
  {sparse: true})
```

也可以使用下面的索引来高效地搜索磁盘字段：

```
db.products.createIndex({"attrs.rotation": 1}, {sparse: true})
```

这里总体的策略就是为了可读性和应用发现性来划分字段特性的范围，并且使用稀疏索引来

排除null值。

如果字段特性完全无法预测，就无法创建单独的索引。这就必须使用不同的策略——此时，为名称和值对的数组——如下面例子所示：

```
{ _id: ObjectId("4d669c225d3a52568ce07646")
  sku: "ebd-123"
  name: "Hi-Fi Earbuds",
  type: "Headphone",
  attrs: [ {n: "color", v: "silver"},
            {n: "freq_low", v: 20},
            {n: "freq_hi", v: 22000},
            {n: "weight", v: 0.5}
          ]
}
```

attrs指向了一个子文档数组。每个文档都有2个值n和v，对应每个动态特性的名字和值。这个范式表示允许我们使用单个组合索引来为这些特性建立索引：

```
db.products.createIndex({"attrs.n": 1, "attrs.v": 1})
```

可以使用这些特性字段进行查询，但是要这样做，必须使用\$elemMatch操作符：

```
db.products.find({attrs: {$elemMatch: {n: "color", v: "silver"}}})
```

注意，这个策略会产生很多开销，因为它需要在索引里存储键的名字。在生产环境部署之前，最好使用代表性数据来测试一下性能。

B.7 事务

Transactions

MongoDB没有为一组操作提供ACID保证（但是单个的操作是原子性的），并且没有与RDBMS对应的BEGIN、COMMIT、ROLLBACK语义。当需要这些功能时，使用不同的数据库（或者数据需要恰当的事务或者应用整体上需要）或者不同的设计一个接一个来执行一系列操作。MongoDB仍然支持原子性、单独文档的持久化更新和一致性读，而且这些特性根本上可以在单个应用中用来实现类事务行为。

我们在第6章看了订单授权和仓库管理的例子。本节里实现的工作队列修改一下就可以支持回滚。两种情况下，类事务行为的基础是findAndModify命令，它用来原子性操作文档的state字段。

所有这些情况里使用的事务性策略都可以描述为compensationdriven^[1]。

^[1]关于补偿驱动事务的两篇文章值得研究。最初的是 Garcia-Molina 和 Salem 的《Sagas》论文 (<http://mng.bz/73is>)。另外一篇非正式但是非常有趣的文章 Gregor Hohpe 撰写的《Your Coffee Shop Doesn't Use Two-Phase Commit》(<http://mng.bz/kpAq>)也非常值得阅读。

抽象补偿过程是这样的：

- (1) 原子性修改文档的状态。
- (2) 执行一些工作单元，它们可能包括原子性修改其他文档。
- (3) 确保系统整体（所有涉及的文档）都是有效的状态。如果这样，标记事务完成；否则恢复所有文档为之前的状态。

值得注意的是，补偿驱动的策略对于长期多步骤事务是必须的。授权、快递和取消订单是一个例子。这些情况下，RDBMS关系型数据库的事务语义必须实现类似的策略。

或许没有办法避免应用的多对象ACID事务的需求。但是使用正确的模式，MongoDB可以分担一些事务工作并支持应用需要的事务性语义机制。

B.8 定位与预计算

Locality and precomputation

经常用MongoDB作为分析数据库，许多用户在MongoDB里存储分析数据。原子性增长和丰富文档的组合看起来是最棒的。例如，每天页面访问总量的文档汇总起来就可以作为月浏览量。为了简单，下面的文档包含了前5天的页面浏览总数：

```
{ base: "org.mongodb", path: "/",
  total: 99234,
  days: {
    "1": 4500,
    "2": 4324,
    "3": 2700,
    "4": 2300,
    "5": 0
  }
}
```

我们可以直接使用简单的\$inc操作符来更新按“天”和按“月”的浏览总数：

```
use stats-2011
db.sites-nov.update({ base: "org.mongodb", path: "/" },
  $inc: {total: 1, "days.5": 1});
```

花点时间来看看集合和数据库的名字。集合sitesnov定位到特定的月份上，而数据库名字stats-2011包含年份。这为应用提供了良好的定位性。当查询最新的访问时，只需要查询相关的集合，与总体的历史记录做对比即可。如果需要删除数据，就可以删除某个时间范围内的集合，而不是从超大集合里删除部分文档。后者可能导致磁盘碎片。这里用的其他原则是预计算。有时候靠近每月开始的时间，可以使用0初始化每天的文档。结果，以后增加计数器也不会修改文档的大小，因为它们实际上每月增加新的字段，而我们只会修改它们的值。这一点非常重要，因为它避免了在磁盘上搬迁文档。搬迁速度很慢，并且经常

会导致磁盘碎片。

B.9 反模式

Antipatterns

MongoDB缺少约束，这可能导致糟糕的数据组织。以下是一些生产环境部署常见的问题。

B.9.1 粗心索引

当用户遇到性能问题时，通常不难发现未使用和低效率的索引。应用程序的高效索引通常会基于查询分析进行设计。请参考第7章介绍的优化方法。

记住，当有不需要的索引时，插入和更新会花费更长的时间，因为必须更新关联的索引。根据经验，应该定期检查索引以确保使用了正确的索引做正确的工作。

B.9.2 交错类型

确保集合中相同名字的键的类型是相同的。如果存储了手机号，例如，一致性存储，要么是字符串要么是整数（但是不能混合两种类型）。单个键值里混合类型会导致应用的逻辑过于复杂，而且难以转换BSON文档为强类型实体。

B.9.3 单一集合

集合应该只存储一种类型的实体，不要让商品和用户存在于一个集合中。其实集合非常廉价，每个类型都应该有自己的集合。否则会有并发问题。

B.9.4 大型、深嵌文档

关于MongoDB的文档数据模型还有2个误解。第一个误解是，我们不应该在集合之间建立关系，但是选择在相同的文档里表示管理。这往往导致混乱不堪，但是用户还是尝试这样做。第二个误解是，对于文档一次性地过度解释。文档，这些用户的解释是，一个单独的实体，像真实的文档一样。这导致难以查询和更新大文档，更加难以理解。

这里的底线是，我们应该保证文档尽量地小（每个100 KB以下，除非是存储二进制数据），并且不应该嵌套太深的层次。小的文档让文档更新更加廉价，因为这时候需要在磁盘上完全重写文档时，重写得更少。文档的优势是可理解性，方便开发者设计数据模型。

另外一个好的实践经验是把二进制数据存入单独的集合里，并且使用_id引用。附录C里有粗

粒度二进制数据的内容。

B.9.5 一个用户一个集合

为每个用户建立一个集合确实很少见。一个原因是命名空间（索引加上集合）大概只有24 000个，一旦增长超过最大值，就必须分配新的数据库。另外，每个集合还有索引会带来额外的开销，使得这个策略非常浪费空间。

B.9.6 不可分片集合

如果期望集合增加到足够分片的程度，就要确保最后可以分片。如果可以给集合定义一个高效的分片键，集合就是可分片的。可以参照第12章的选择分片键值的技巧提示。

C.1.2 存储 MD5

把数据存储在二进制格式时，数据会以二进制形式存储。二进制数据在存储时，每个字节都会被存储为一个字节。二进制数据在存储时，每个字节都会被存储为一个字节。

BSON包含一个二进制数据类型，称为二进制数据类型。二进制数据类型在存储时，每个字节都会被存储为一个字节。二进制数据类型在存储时，每个字节都会被存储为一个字节。

注意：当创建BSON二进制数据类型时，使用二进制数据类型。二进制数据类型在存储时，每个字节都会被存储为一个字节。二进制数据类型在存储时，每个字节都会被存储为一个字节。

二进制数据类型在存储时，每个字节都会被存储为一个字节。二进制数据类型在存储时，每个字节都会被存储为一个字节。

二进制数据类型在存储时，每个字节都会被存储为一个字节。二进制数据类型在存储时，每个字节都会被存储为一个字节。

二进制数据和网格文件系统

Binary data and GridFS

对于存储图片、缩略图、音频等二进制文件，许多应用只依赖于文件系统。虽然文件系统提供了更快的访问方式，但是文件存储可能会导致混乱。思考下文件系统在每个文件夹下的最大限制。如果有几百万的文件要去追踪，就需要分割到多个文件夹分离存储。另外的困难是元数据。因为文件元数据仍然存储在数据库里，所以执行精确的文件和元数据备份变得异常复杂。

对于特定的使用情况，在数据库中存储文件可能更合理，因为这样会简化文件的组织和备份工作。在MongoDB里，可以使用BSON二进制文件来存储任意类型的二进制数据。这个数据类型对应于RDBMS BLOB（二进制大对象）类型，而且它是MongoDB提供的两种二进制对象存储机制的基石。

每个文件一个文档的存储方式对于小的二进制对象是最佳的方式。如果需要分类管理大量的缩略图或者二进制MD5文件，则使用单个文档存储更加简单。换句话说，有时候可能要存储大的图片或者音频、视频文件。此时，GridFS，用户存储任意大小二进制文件的MongoDB API，是更好的选择。接下来两节我们会看到两种方式的完整例子。

C.1 简单二进制存储

Simple binary storage

BSON包含了一个一级类型来存储二进制数据。我们可以使用它在MongoDB文档里存储二进制对象。对象大小的唯一限制就是文档大小的限制，MongoDB 2.0之后的文档限制是16 MB。因为大的文档会消耗系统资源，所以鼓励使用GridFS来存储大于1 MB的任意二进制对象数据。

我们将会看一下两个在单个文档里存储二进制对象的合理的使用。

首先，我们会看一下如何存储缩略图，然后我们会看看如何存储MD5。

C.1.1 存储缩略图

假设我们要存储缩略图的集合，则代码很简单。

首先获取文件名，canyon-thumb.jpg，然后读取数据到本地变量。接下来使用BSON二进制对象来包装原始二进制数据，Ruby驱动的BSON::Binary构造函数：

```
require 'rubygems'
require 'mongo'
image_filename = File.join(File.dirname(__FILE__), "canyon-thumb.jpg")
image_data = File.open(image_filename).read
bson_image_data = BSON::Binary.new(image_data)
```

剩下的工作就是构建一个简单的文档来存放二进制数据，然后插入数据库中：

```
doc = {"name" => "monument-thumb.jpg",
      "data" => bson_image_data }
@con = Mongo::Client.new(['localhost:27017'], :database => 'images')
@thumbnails = @con[:thumbnails]
result = @thumbnails.insert_one(doc)
```

提取二进制数据，获取文档。在Ruby里，用to_s方法把数据解包为二进制字符串，我们可以使它来与最初的数据比较：

```
@thumbnails.find({"name" => "monument-thumb.jpg"}).each do |doc|
  if image_data == doc["data"].to_s
    puts "Stored image is equal to the original file!"
  end
end
```

如果运行前面的脚本，就会看到提示信息：2个文件是相等的。

C.1.2 存储 MD5

把校验码存储为二进制数据非常常见，这是二进制BSON类型的另外一个潜在用途。以下是如何生成缩略图的MD5值，并存储到文档里：

```
require 'md5'
md5 = Digest::MD5.file(image_filename).digest
bson_md5 = BSON::Binary.new(md5, :md5)
@thumbnails.update_one({:_id => @image_id}, {"$set" => {:md5 => bson_md5}})
```

注意，当创建BSON二进制对象时，使用:md5标签。另外一个额外的字段用来表示存储的是什么二进制数据。这个字段是可选的，不影响数据库如何存储或者解释数据^[1]。

^[1]这还不是技术。默认子类型 2 表示附加的二进制数据也包含 4 个额外的字节来表示大小，这确实会影响一些数据库命令。当前默认的子类型是 0，并且所有的子类型现在存储的二进制数据方式是一样的。子类型因此可以看做一种轻量级的标签，可以被应用开发者使用。

查询刚才保存的文档非常简单。但是注意，为保证返回文档尽可能小和可读排除了字段：

```
> use images
> db.thumbnails.findOne({}, {data: 0})
{
  "_id" : ObjectId("4d608614238d3b4ade000001"),
  "md5" : BinData(5,"Klud3EUjT49wdMdkOGjbDg=="),
  "name" : "monument-thumb.jpg"
}
```

查看MD5字段，明显标记为二进制数据（表5.6中介绍），使用了子类型和原生数据。记住，MongoDB首先是根据长度或数据大小排序BinData，再次根据BSON的一个字节的子类型，最后才是根据数据，逐字节进行比较。

C.2 网格文件

GridFS

GridFS是MongoDB数据库存储任意大小文件的机制。GridFS规范被所有官方驱动和MongoDB的mongofiles工具支持，确保跨平台的一致性访问。GridFS对于存储大的二进制对象非常有用。它足够快速满处理这些对象，统称的存储方法借助了streaming流处理机制^[1]。

词语GridFS可能给大家带来迷惑，所以有两点必须澄清。首先GridFS特性并非MongoDB固有。正如之前提到的，它是所有官方驱动管理大的二进制对象的约定convention。其次，GridFS并没有丰富的真正的文件系统的语义。例如，没有锁定和并发的协议，而且GridFS的接口限制在简单的保存、查询和删除操作上。这意味着如果要更新文件，就需要删除它，然后添加新的版本。

GridFS通过将大的文件分割为较小的225KB数据块实现文件存储，然后把每个数据块保存为单独的文档——MongoDB 2.4.10之前的版本使用256 KB的数据块。默认情况下，这些数据块保存到名为fs.chunks的集合中。一旦写完这些数据块，文件的元数据就被存储到fs.files集合中的单个文档里。图C.1包含了简单的演示说明如何保存1 MB 大小的文件canyon.jpg。注意：这里GridFS中的数据块和分片集群中的数据块无关。

使用GridFS^[2]的理论知识已经足够了。接下来，我们将会实战如何通过Ruby GridFS API 和 mongofiles 工具来实现GridFS功能。

^[1]【译者注】流处理，我在高级架构班讲解大数据存储与优化传输机制的时候也介绍了这个概念，包括hadoop的大数据存储都使用了流处理机制。WCF等一些优秀的分布式框架也提供了支持。

^[2]我们可以在 <http://docs.mongodb.org/manual/core/gridfs/> 和 <http://docs.mongodb.org/manual/reference/gridfs/> 查询更多有关GridFS的知识。

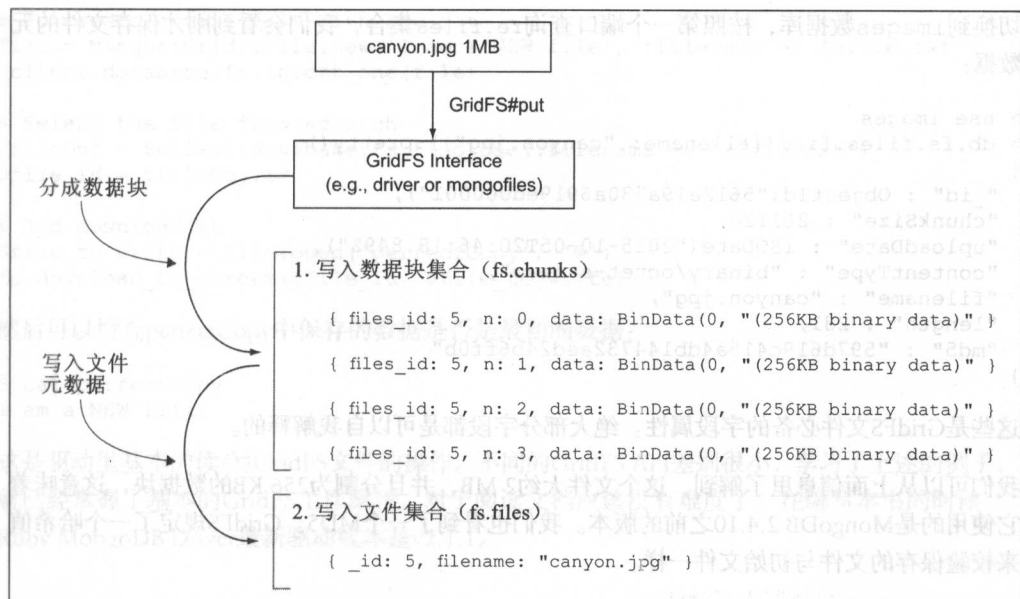


图 C.1 Storing a file with GridFS using 256 KB chunks on a MongoDB server prior to v2.4.10

C.2.1 Ruby 中的 GridFS

前面存储的是小的缩略图。缩略图只占用10KB空间，因此可以单独保存在单个文档里。最初的图片几乎是2MB大小，因此更适合GridFS存储。这里我们来使用Ruby的GridFS API来存储原始的文件。首先，连接数据库，然后初始化Grid对象。这里会保存数据库GridFS文件的引用。

接下来，打开原始的图片文件canyon.jpg。GridFS最基本的接口就是保存和查询。这里使用Grid#put方法保存，它接受二进制数据字符串或者I/O对象，比如文件指针。传递文件指针，将数据保存到数据库。

使用最新的Ruby MongoDB驱动，这个方法会返回文件的唯一对象ID：

```

require 'rubygems'
require 'mongo'
include Mongo
$client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'images')
fs = $client.database.fs
$file = File.open("canyon.jpg")
$file_id = fs.upload_from_stream("canyon.jpg", $file)
$file.close
  
```

正如介绍的，GridFS使用了2个集合来存储文件数据。第一个叫做fs.files，保存每个文件的元数据。第二个是fs.chunks，保存每个文件的数据块。我们来简单查看一下这些信息。

切换到images数据库，按照第一个端口查询fe.files集合，我们会看到刚才保存文件的元数据：

```
> use images
> db.fs.files.find({filename: "canyon.jpg"}).pretty()
{
  "_id" : ObjectId("5612e19a530a6919ed000001"),
  "chunkSize" : 261120,
  "uploadDate" : ISODate("2015-10-05T20:46:18.849Z"),
  "contentType" : "binary/octet-stream",
  "filename" : "canyon.jpg",
  "length" : 281,
  "md5" : "597d619c415a4db144732aed24b6ff0b"
}
```

这些是GridFS文件必备的字段属性。绝大部分字段都是可以自我解释的。

我们可以从上面信息里了解到，这个文件大约2 MB，并且分割为256 KB的数据块，这意味着它使用的是MongoDB 2.4.10之前的版本。我们也看到了一个MD5。GridFS规定了一个哈希值来校验保存的文件与初始文件一样。

每个数据块都在files_id字段了保存了文件的对象ID。因为可以轻易计算出文件的数据块数量：

```
> db.fs.chunks.count({"files_id" : ObjectId("4d606588238d3b4471000001")})
8
```

给定文件和数据块大小，2个数据块就应该够了。很容易查看数据块的内容。正如先前一样，我们想通过排除数据来保证输出信息的可读性。这个查询会返回最初的8个数据块，正如n的值表示的：

```
> db.fs.chunks.findOne({files_id: ObjectId("4d606588238d3b4471000001")},
  {data: 0})
{
  "_id" : ObjectId("4d606588238d3b4471000002"),
  "n" : 0,
  "files_id" : ObjectId("4d606588238d3b4471000001")
}
```

读取GridFS文件和写文件一样简单。下面的例子将创建一个文本文件，设置名字，然后使用GridFS保存。接下来可以使用find_one()语句来查询它，返回的是Mongo::Grid::File对象。然后从Mongo::Grid::File对象里获取文件ID，再使用这个ID从数据库里查询文本文件，这个文件是使用perfectCopy名字来保存的：

```
require 'rubygems'
require 'mongo'
include Mongo

$client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'garden')
fs = $client.database.fs
```

```
# To create a text file with raw data
file = Mongo::Grid::File.new('I am a NEW file', :filename => 'aFile.txt')
$client.database.fs.insert_one(file)

# Select the file from scratch
$fileObj = $client.database.fs.find_one(:filename => 'aFile.txt')
$file_id = $fileObj._id

# And download it
$file_to_write = File.open('perfectCopy', 'w')
fs.download_to_stream($file_id, $file_to_write)
```

然后可以检查perfectCopy中保存的数据是否是最初的数据：

```
$ cat perfectCopy
I am a NEW file!
```

这是驱动里基本的读/写GridFS文件的操作。不同的GridFS API差别很小，学习了上述的例子，就已经掌握了基本的GridFS工作原理，对于阅读文档应该没有难度了。在编写本书的时候，Ruby MongoDB Driver最新驱动版本是v2.1.1。

C.2.2 使用 mongofiles 操作 GridFS

可以使用命令来列表、保存、查询和删除GridFS文件。例如，我们可以列出数据库中所有的images文件：

```
$ mongofiles -d images list
connected to: 127.0.0.1
canyon.jpg 2004828
```

添加文件也很方便。以下是如何使用Ruby脚本来添加图片的拷贝数据：

```
$ mongofiles -d images put canyon-copy.jpg
connected to: 127.0.0.1
added file: { _id: ObjectId('4d61783326758d4e6727228f'),
  filename: "canyon-copy.jpg",
  chunkSize: 262144, uploadDate: new Date(1298233395296),
  md5: "9725ad463b646ccbd287be87cb9b1f6e", length: 2004828 }
```

我们也可以列出文件来检验拷贝已经写入了：

```
$ mongofiles -d images list
connected to: 127.0.0.1
canyon.jpg 2004828
canyon-copy.jpg 2004828
```

mongofiles支持许多参数，我们可以使用-help参数来查看所有参数：

```
$ mongofiles --help
Usage:
  mongofiles <options> <command> <filename or _id>
```

Manipulate gridfs files using the command line.

Possible commands include:

- list - list all files; 'filename' is an optional prefix which listed filenames must begin with
- search - search all files; 'filename' is a substring which listed filenames must contain
- put - add a file with filename 'filename'
- get - get a file with filename 'filename'
- get_id - get a file with the given '_id'
- delete - delete all files with filename 'filename'
- delete_id - delete a file with the given '_id'

See <http://docs.mongodb.org/manual/reference/program/mongofiles/> for more information.

general options:

- help print usage
- version print the tool version and exit

verbosity options:

- v, --verbose more detailed log output (include multiple times for more verbosity, e.g. -vvvvv)
- quiet hide all log output

connection options:

- h, --host= mongodb host to connect to (setname/host1,host2 for replica sets)
- port= server port (can also use --host hostname:port)

authentication options:

- u, --username= username for authentication
- p, --password= password for authentication
- authenticationDatabase= database that holds the user's credentials
- authenticationMechanism= authentication mechanism to use

storage options:

- d, --db= database to use (default is 'test')
- l, --local= local filename for put/get
- t, --type= content/MIME type for put (optional)
- r, --replace remove other files with same name after put
- prefix= GridFS prefix to use (default is 'fs')
- writeConcern= write concern options e.g. --writeConcern majority, --writeConcern '{w: 3, wtimeout: 500, fsync: true, j: true}' (defaults to 'majority')

MongoDB 实战 (第二版)

MongoDB面向文档的数据库完全支持高可用集群、动态数据架构,可以轻易实现跨越多台服务器的分布式数据存储。MongoDB 3.0即使在大数据负载压力下依然可以保持灵活、伸缩和快速的特性。

《MongoDB实战》(第二版)引入了MongoDB 3.0和面向文档的数据模型。本书为系统开发工程师和系统架构工程师提供了从宏观架构到底层细节的详细内容。本书包含的许多例子可以帮助大家进行关键领域的数据建模。读者会享受阅读本书管理主从复制、自动分片和集群部署的深入内容。

内容包含:

- 索引、查询和标准DB操作
- 聚合与文本搜索
- 实现自定义聚合和报告查询
- 高可用和伸缩集群部署
- 更新到Mongo v3.0以后

写给MongoDB工程师和高级开发人员

Kyle Banker在离开MongoDB开发团队之后,就职于一家创业公司。

Peter Bakkum就职于MongoDB企业版开发团队。

Shaun Verch就职于MongoDB核心服务器团队。

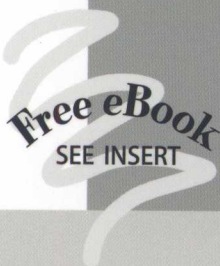
Doug Garrett是MongoDB大数据分析创新大奖的获得者;软件架构师。

Tim Hawkins是Yahoo欧洲搜索工程部门的领导。

技术贡献: Wouter Thielen
技术编辑: Mihalis Tsoukalos

策划编辑: 谢燕群
责任编辑: 谢燕群

上架建议: 数据库>NoSQL高级开发



学习、实践和开发MongoDB的完美手册。

——Jeet Marwah, Acer Inc.

MongoDB数据库开发和数据建模学习的最佳必读书籍。

——Hernan Garcia, Betterez Inc.

提供了MongoDB学习必备的详细知识。

——Gregor Zurowski,
独立软件开发顾问

太棒了, MongoDB的精华知识。

——Hardy Ferentschik, Red Hat

ISBN 978-7-5680-2579-9



9 787568 025799 >

定价: 80.00元